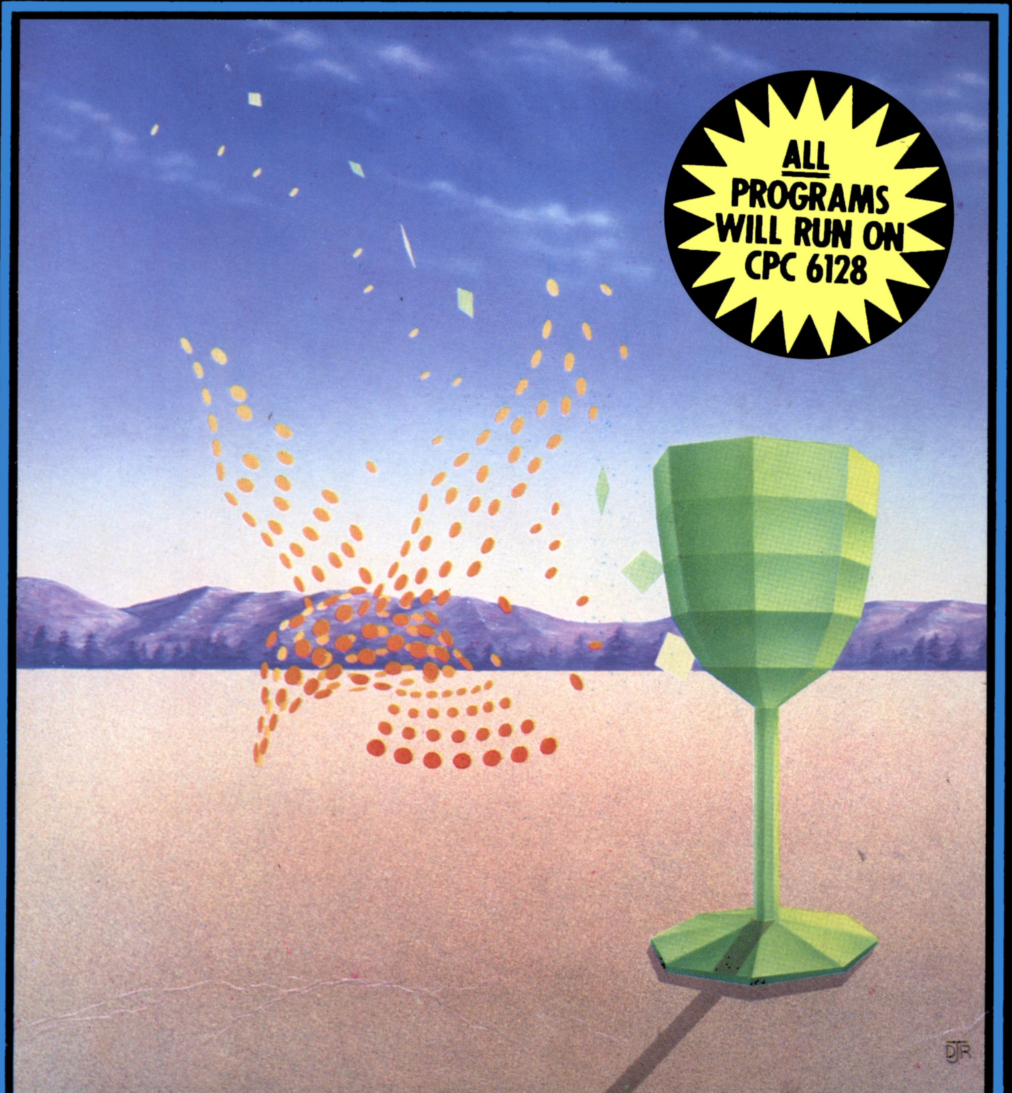
ARGUS BOOKS

WORKING GRAPHICS ON THE AMSTRAD CPC 464 & 664

MIKE JAMES, KAY EWBANK & S.M.GEE



Working graphics on the AMSTRAD CPC464 & 664

Working graphics on the AMSTRAD CPC 464 & 664

Mike James, Kay Ewbank
& S.M. Gee

ARGUS BOOKS

Argus Books Limited
1 Golden Square
London W1R 3AB

ISBN 0 85242 874 X

Phototypesetting by Photocomp Ltd, Birmingham

Printed and bound by Whitstable Litho

Preface

Computer graphics is a subject that has an immediate appeal, but producing the effects that are so impressive in computer games, simulations and screen displays (such as those to be seen on television and on films) takes more than just an understanding of the Amstrad's graphics commands. The purpose of this book is to show how, using nothing more than BASIC, the Amstrad CPC464, 664 and 6128 can be used to produce high quality graphics – a glance through the illustrations will quickly indicate both the scope of the book and the results that you can expect to achieve with it.

The first chapter presents the Amstrad's BASIC graphics commands in a logical fashion with the emphasis on how they are used rather than just what they do. After this first chapter the rest of the book concentrates on three topics – animation, painting and two and three dimensional point and line graphics. The theory behind each topic is explained and illustrated using programs that range from small demonstrations to complete applications. If you are looking for example programs then you will find plenty to interest you – both by way of games and recreation and serious applications. Amongst others are an animated game, a painting and drawing program, a two dimensional draughting program and a three dimensional viewer.

All of the programs in this book have been reproduced directly from listings of working programs and so we are confident that if you type them in as presented they will work. You should find the programs easy to understand because they have been developed using a natural structured programming style. None of the programs have been renumbered, because while this would make entering the programs easier it would mean losing the line numbering scheme that we used while developing the programs. If you wish to modify or extend any of the programs you will find the explanations of the subroutine structure a handy reference.

Thanks are due to Phil Chapman for turning our words into a book.

Contents

Amstrad graphics 1

No matter what you are interested in using your Amstrad for, graphics is bound to be an important feature. In its own right graphics is an interesting subject, but it is also fundamental to almost every application. This book sets out to show just how much can be achieved using the Amstrad's graphics capabilities. Later chapters explain and demonstrate animation, graphs and charts, computer assisted painting and 2D and 3D drawing. Some of the program examples given later are fully working, ready-to-use applications programs (in particular see AMART in Chapter Five, AMPLOT in Chapter Six and AMVIEW in Chapter Seven). Even though the emphasis is on the practical uses of graphics, there are explanations which will enable you to understand graphics and go on to modify the examples or produce your own programs.

First, you need to be able to program in BASIC and, of course, to be familiar with your Amstrad. This chapter describes the use of the Amstrad's BASIC graphics commands and, if you are at all confused or uncertain about how these commands work, read it carefully before moving on. If you feel confident that you already understand the commands, it is still worth looking through this chapter as a refresher. To get the most out of this book it is important to try the examples. Very often it is only by experimenting with something that the real power of a method becomes clear.

Finally, it is worth saying that all of the program listings in this book have been produced directly from the Amstrad, thus avoiding the errors of typesetting. All of the 'printer dumped' illustrations were produced using the programs given in this book and hence they represent what you can expect to achieve yourself. In fact, since many of the programs depend on colour or animation you should be able to do better!

All of the programs will run (and have been tested) on both the

Amstrad CPC 464 and the 664. Apart from the few extra graphics commands which the 664 has, which are described separately, all of the discussion of the methods and programs apply equally to the 464 and the 664. The high level of compatibility of the CPC 6128 with its smaller relatives means that the programs in this book should also run on the 6128.

The pixel and display modes

All computer graphics is based on the idea of a display made up of 'pixels' (short for 'picture elements'). A pixel is simply the smallest area of the display that can be set to a given colour. The size and number of pixels in a display governs the fineness of detail that can be represented, and so the more pixels the better. However, the colour of each pixel has to be stored in memory, and more pixels use more memory. In practice a compromise is struck between memory saved and the number of pixels.

Although all computer graphics is achieved by setting the colour of individual pixels it is not always convenient to work one pixel at a time. In particular when it comes to printing text it would be very tedious to have to specify the colour of each pixel used to form a letter 'A' for example. To overcome this problem commands are made available that automatically set groups of pixels to form the shapes of any given letter in a single operation – this is referred to as 'character graphics' and it corresponds to the well known idea of PRINTing text to the screen. What is important to notice here is that PRINTing characters to the screen uses the same display mechanism as the more obviously 'graphics' operations such as drawing a line – i.e. setting a group of pixels to a specified colour. (This is not the case with some other machines, where printing text and graphics have to be done in different display modes.)

The Amstrad has three different display modes that can be used for printing text, or for graphics, or any combination of the two. Each mode offers the Amstrad programmer a different total number of pixels and number of colours. In terms of text display this amounts to a different number of characters on a line, lines on a screen and number of colours. Each mode is, in fact, a compromise between spatial resolution (i.e. number of pixels) and colour resolution (i.e. how many colours can be used). The details of the three modes are given in the following table.

Mode	*characters by lines*	*horizontal by vertical*	*no of colours*
0	20x25	160x200	16
1	40x25	320x200	4
2	80x25	640x200	2

The column labelled 'horizontal by vertical' gives the size of the screen in terms of the pixels that make up the screen, and this will be discussed more fully later in the chapter. No matter what mode you are in, a total of 16K of memory is used to store the screen. There is, however, still plenty of memory left over for programs. It is worth noticing that altering the display mode doesn't alter the horizontal resolution which is fixed at 200 pixels.

To change mode use
MODE m
where 'm' is the number of the mode required.

Character and pixel graphics

In any of the Amstrad's graphics modes there are two possible ways of working. The first is based on using commands that manipulate characters, and the second is based on using commands that are concerned with pixels. Character based graphics is generally easy to use and fast, and for this reason it is often used for animation. Pixel based or 'high resolution' graphics is used for drawing outlines, graphs, backgrounds and, of course, everything that is difficult using character graphics! This division is admittedly artificial but it does make sense to deal first with both type of graphics in turn and then look at the ways in which they can be made to work together.

Character graphics

PRINT and LOCATE

To be able to create graphics displays using characters it is necessary to have some way of positioning output anywhere on the screen and either a good range of graphics characters or the ability to define new graphics characters, or both. The first requirement is satisfied by the LOCATE command. LOCATE x,y will cause the next output to appear at line y and column x. The printing positions are counted from the left hand side of the screen starting with column 1 and finishing with column 20 in mode 0, 40 in mode 1 and 80 in mode 2. Similarly, the line numbering goes from 1, for the top line, to 25, for the bottom line. Thus to print a letter A for example at line y and column x use

LOCATE x,y: PRINT "A";

The trailing semi-colon at the end of the PRINT statement suppresses the automatic line feed that is normally performed at the end of every PRINT statement. The only time that an automatic line feed after a PRINT causes a problem in graphics is when it causes the screen to scroll. If you would like to see the effect of leaving the semi-colon off the end of the PRINT statement try

```
10 MODE 2
20 LOCATE 80,25
30 PRINT "A"
40 GOTO 20
```

This program should repeatedly print the letter A at the same position on the screen but, because this position is the bottom right hand corner, the line feed causes the screen to scroll and hence produces a 'stream' of letter As. If you add a semicolon onto the end of line 30 you will see only a single letter A as intended.

When you are typing something in to the Amstrad the position where the next character will appear is marked by a block called the 'text cursor'. Even though the text cursor is not displayed while a BASIC program is running you can still imagine that LOCATE x,y has the effect of moving the text cursor to row y and column x.

The character set – ASC and CHR$

The second requirement for character graphics is a good range of character shapes. The Amstrad has rather more than just the A to Z and a to z characters that are found on most computers. In order to make the large collection of characters easier to use each one is assigned a code number – its 'ASCII code'. For example the letter 'A' has an ASCII code of 65 and 'a' has an ASCII code of 97.

The full set of ASCII codes can be found in Appendix III of the 'User Instructions'. However you can also use the function ASC to discover the ASCII code of any particular character as ASC(A$) will return the ASCII code of the first character in the string A$. In the same way you can find out which character corresponds to any ASCII code using the CHR$ function, as CHR$(n) returns the single character with ASCII code n. Notice that ASC takes you from the letter to the code and CHR$ from the code to the letter. It is also important to know that, although the ASCII code starts at 0 and ends at 255, the characters from 0 to 31 are special in that they do not correspond to anything that can be printed on the screen – they are non-printable or 'control' chraracters and they are described later. To see the complete character set use

```
10 MODE 1
20 FOR I=32 to 255
30  PRINT CHR$(I);
40 NEXT I
```

Notice that by starting the FOR loop at 32 we have avoided trying to print the control characters.

User-defined graphics characters – SYMBOL and SYMBOL AFTER

Although the standard character set described above is fairly comprehensive there is always the possibility that the character or characters that you need are missing. Fortunately there is no need to worry – the entire Amstrad character set can be redefined to make any shape correspond to any ASCII code. Before we explain how to define new characters we first have to examine how characters are produced on the screen. Every character that the Amstrad can display on the screen is produced from a grid of 64 dots arranged into a square, eight dots by eight dots. The pattern of any character depends on which dots in the grid are displayed as black and which are displayed as white. (How colour is handled is best described later; for the moment it is simpler to think about how the machine works when it is first switched on – i.e. bright letters on a dark background.) The black dots are referred to as 'background' dots because it is usual to construct character shapes as a pattern of white dots using the black dots as a background. For the same reason the white dots are usually called 'foreground' dots. For example, the letter 'a' can be produced by the pattern of dots shown in fig 1.1 (f stands for foreground and b for background).

```
bbbbbbbb
bbbbbbbb
bbfffbbb
bbbbbfbb
bbffffbb
bfbbbfbb
bbffffbb
bbbbbbbb
```

Fig 1.1 User-defined graphics – foreground and background dots

You might find it difficult to see the pattern of the letter 'a' among the 'f's' and 'b's' but it becomes very clear if each 'f' is replaced by an asterisk and each 'b' is replaced by a blank as in fig 1.2.

```
  ***
     *
  ****
 *   *
  ****
```

Fig 1.2 User-defined letter 'a'

Obviously, if we are going to define the shape that corresponds to a user-defined graphics character, there must be some way of specifying which dots in the 8 by 8 grid are foreground and which are background. The definition of the new character has to be done a row at a time. Each row in the grid can be written as a sequence of eight digits by writing a 0 for every background dot and a 1 for every foreground dot. For example, the row 'bbfffbbb' would be written as '00111000'. This row of eight noughts and ones has to be converted to a single decimal number by multiplying each one by a weight and adding up the result. The weights for each nought or one starting from the far left are

128 64 32 16 8 4 2 1

which gives

0*128+0*64+0*32+1*16+1*8+0*4+0*2+0*1

or 24 for the row of dots given by

0 0 0 1 1 0 0 0

To produce the definition of a complete character you have to convert each of the eight rows of dots to eight numbers that represent the shape of the character. Once you have done this you can use the statement

```
SYMBOL x,r1,r2,r3,r4,r5,r6,r7,r8
```

to define the character whose ASCII code is 'x', where r1 to r8 are the eight numbers that represent the eight dot rows. The Amstrad sets aside enough memory to store the definitions of characters corresponding to the ASCII codes 240 to 255, that is 16 characters. It is possible to reserve more memory for more character definitions but it is rare that more than 16 user-defined graphics characters are needed in the same program. If you do need more user-defined characters then simply use

```
SYMBOL AFTER x
```

which will make all the characters corresponding to ASCII codes x to 255 to be user-definable. A second effect of a SYMBOL AFTER statement is that it resets all of the character definitions back to what they were when the machine was first switched on.

Converting each dot row of the letter 'a' pattern given above results in the following SYMBOL statement

```
SYMBOL 240,0,0,56,4,60,68,60,0
```

which sets the definition of character 240, that is following this SYMBOL statement PRINT CHR$(240) will print the letter 'a'.

Hexadecimal numbers – HEX$ and BIN$

Although the coding of each row of dots in a user-defined character into a single number has been explained in terms of the familiar decimal numbers and arithmetic it is a lot easier to use hexadecimal numbers. The reason for this is simply that the row of zeros and ones that represent the dot pattern is nothing more than a binary number and it is particularly easy to convert a binary number to a hexadecimal number. All that you have to do is to divide the number up into groups of four bits (i.e. four ones or zeros) and convert each group to hexadecimal using the following table

bits	hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

You will notice from this table that a hexadecimal number can contain not only the usual digits 0 to 9 but the letters A to F as well! The Amstrad allows hexadecimal numbers to be used anywhere that a decimal number can. They are distinguished from decimal numbers by the use of the prefix '&'. So, in Amstrad BASIC, 3 is a decimal number but &3 is a hexadecimal number. For example the row of dots corresponding to 00011000 can be converted to a hexadecimal number by looking up the two groups of four bits. The first group of four bits is 0001 and this corresponds to 1 in the table. The second group of four bits is 1000 and this corresponds to 8 in the table, making the hexadecimal number that represents 00011000 equal to &18. As another example the hexadecimal equivalent of 11101000 is &E8, that is

```
  1110  1000
& E       8
```

Using hex numbers the definition of the letter 'a' given earlier becomes

```
SYMBOL 240,&00,&00,&38,&04,&3C,&44,&3C,&00
```

If you find the above conversion method difficult to apply, then it will come as something of a relief to discover that the Amstrad has a function that will convert numbers to hex for you. The function HEX$(n) will return a string that is the hex representation of the number n. Thus if you want to know what n is in hex use

```
PRINT HEX$(n)
```

The Amstrad can handle binary numbers in the same way that it can handle hex. If you want to write a binary number in Amstrad BASIC simply write &X in front of it. So 101 is one hundred and one but &X101 is binary 101 or 5 in decimal. There is also a function BIN$(n) which will return a string that is the binary representation of n.

Using all this information it is easy to see that to convert to hex a line of zeros and ones in a character definition all you have to use is

```
PRINT HEX$(&Xb)
```

where b is a sequence of eight zeros or ones. For example

```
PRINT HEX$(&X00111000)
```

will print 38 as the hex value for the third line in the definition of the letter 'a' given above.

Controlling colour – PAPER, PEN and INK

The Amstrad is capable of producing 27 different colours ranging from black to bright white. However as the maximum number of colours that can be used at any time is 16 (in mode 0) there obviously has to be some way of selecting which of the 27 possible colours will be used. Each is indicated by a code number between 0 and 26, which never changes. So, for example, orange is always colour number 15. These colour numbers can be referred to as 'physical colour codes' because each number corresponds to a definite colour. On the other hand, the colours that are available in any given mode are also referred to by code numbers – i.e. 0 to 15 in mode 0, 0 to 3 in mode 1 and 0 and 1 in mode 2 and these colour codes do not always correspond to the same physical colour. For this reason they can be called 'logical colours'. A logical colour is assigned a definite physical colour by the INK command

```
INK L,P
```

where L is the number of the logical colour that is assigned to physical colour P. (The Amstrad manual refers to logical colours as 'inks' but, as this tends to be confused with the INK command, we will use the term 'logical colour'.) For example in MODE 2 there are only two logical colours 0 and 1 and the commands

```
INK 0,21
INK 1,15
```

would assign logical colour 0 to physical colour 21, i.e. lime green, and logical colour 1 to physical colour 15, i.e. orange, (giving an orange on lime green display – YUK!!). The important point is that although you are limited in the number of colours you can use in any mode you can select them from the full range of physical colours. A more subtle point is that you can change the logical to physical assignment at any time during a program and hence change the colours that are already displayed on the screen very rapidly. A slightly more advanced form of the INK command is

```
INK L,P1,P2
```

which assigns logical colour L to physical colours P1 and P2 alternately – that is flashing between P1 and P2.

Apart from the INK command the only other command that uses physical colours is

```
BORDER P
```

which sets the border of the display to physical colour P. All the other colour control commands work in terms of the logical colours available in any given mode.

As already explained, there are two colours associated with a character, its foreground colour and its background colour. The colour of foreground pixels is set by the command

```
PEN L
```

and the colour of the background pixels is set by

```
PAPER L
```

where L is a logical colour. For example the letter 'A' can be printed in a number of combinations of foreground and background colours using

```
10 MODE 0
20 FOR L=0 TO 15
30  INK L,L
40 NEXT L
50 FOR L=0 TO 15
60  PEN L
70  PAPER 15-L
80  PRINT "A";
90 NEXT L
```

The first FOR loop in this example sets up the 16 logical colours (0 to 15) to correspond to the first 16 physical colours (also 0 to 15). The second FOR loop sets the pen colour to each logical colour in turn and sets the paper colour to 15-L and then prints 'A'.

This is all there is to controlling colour in character graphics. In every program you should

1 Set the logical to physical colour assignment
2 Select a foreground colour using PEN
3 Select a background colour using PAPER

Pixel graphics

Co-ordinates and graphics increments

The main difference between character and pixel or high resolution graphics is in the use of co-ordinates. The basic method of specifying which pixel a high resolution command refers to is to use x and y co-ordinates which roughly correspond to the columns and lines in character graphics. As the screen is composed of pixels ordered into rows and columns, the x co-ordinate can be thought of as indicating the pixel column and the y co-ordinate the pixel row. The Amstrad will accept co-ordinates as if the screen were composed of 640 pixels horizontally and 400 pixels vertically. Of course the actual number of horizontal pixels varies according to the mode and there are only 200 vertical pixels irrespective of mode so treating the screen as if it was 640 by 400 pixels may seem like an odd thing to do. It does mean, however, that you can write a program in one mode and see what it looks like without having to alter anything apart from the MODE statement. For example, 320,200 is a point in the middle of the screen no matter what mode you are using. However, because there are more co-ordinates than there are pixels some co-ordinates actually refer to the same pixel. For example, in mode 1 there are 320 pixels horizontally and twice as many horizontal co-ordinate values. This means, for example, that the co-ordinate 0,0 refers to the same pixel as 1,0. In this mode if you start off from x,y you have to increase the x co-ordinate by 2 before you can be sure of reaching a new pixel, see fig. 1.3. In the same way, since there are 200 vertical pixels in any mode and twice this number, i.e. 400, of vertical co-ordinates you have to increase y by 2 before you can be sure of reaching a new pixel. This phenomenon is usually referred to as 'aliasing', because each pixel has more than one 'name'. The amount by which you have to increase a co-ordinate before you are sure of reaching a new pixel is a very important quantity, because it is not worth changing a co-ordinate by less than this amount. For example, if you are drawing a straight line in mode 0 by changing the colour of each pixel in turn (an easier way will be explained later) then, if you didn't think carefully, you might use something like

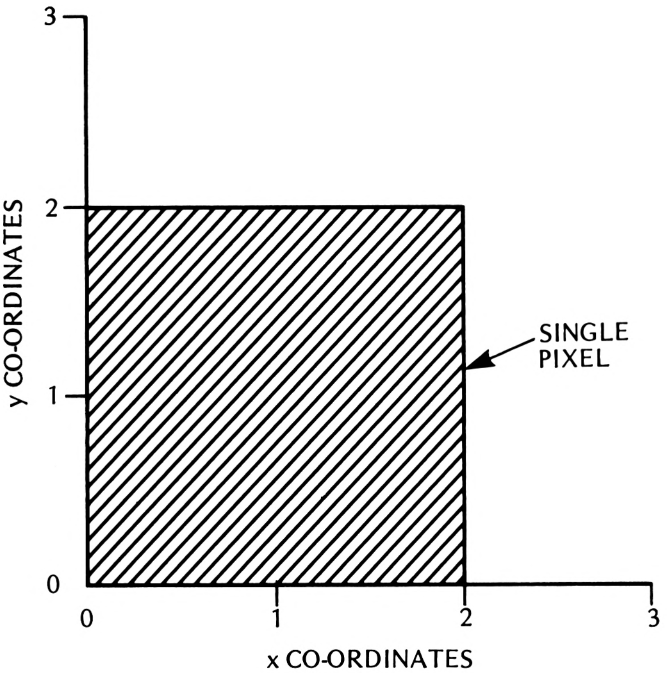


Fig 1.3 A pixel in mode 1

```
10 MODE 0
20 Y=200
30 FOR X=0 TO 639
40 GOSUB 1000
50 NEXT X
```

where, for simplicity, we assume that subroutine 1000 will change the colour of the point at X,Y. The trouble with this program is that it changes the colour of each point no less than FOUR times! It starts off by plotting the pixels at

```
0,200   1,200   2,200   3,200
```

However in mode 0 there are only 160 distinct pixels in any horizontal lines and so all of the above co-ordinates refer to the same pixel. While this program will produce a line it does far more work than is necessary. The correct solution is to change line 30 to

```
30 FOR X=0 TO 639 STEP 4
```

which will plot each pixel only once. In other applications careless use of co-ordinates that results in treating the same pixel more than once can even give an incorrect result, as well as wasting time. The amount by which you have to change a co-ordinate to be sure of reaching a new pixel can be called the 'graphics increment' and it is worth making a table to show its value in each mode

Mode	X *increment*	Y *increment*
0	4	2
1	2	2
2	1	2

In practice, when you are drawing large objects you can forget about the X and Y increments and treat the screen as if it really did have 640 by 400 pixels. However, some of the unexpected results that you might see when using high resolution commands could be the result of trying to use more pixels than are really there!

The graphics cursor

One of the most important ideas in using high resolution graphics is the graphics cursor. In the same way that the text cursor moves around the screen in response to LOCATE, the Amstrad maintains a graphics cursor that moves in response to the high resolution commands. The only real difference is that the graphics cursor isn't made visible by a solid block. However, it is easy to keep track of where the graphics cursor is, because it will always be found at the position of the last pixel that was mentioned in a high resolution graphics command. The current position of the graphics cursor is often made use of in high resolution graphics commands.

The high resolution graphics commands – MOVE, PLOT and DRAW

There are only three high resolution graphics commands and their action is easy to understand. The command

```
MOVE X, Y
```

will move the graphics cursor to X,Y without changing anything on the screen. The command

```
PLOT X, Y
```

will move the graphics cursor to X,Y and change its colour to the current graphics foreground colour and the command

```
DRAW X,Y
```

will move the graphics cursor along a straight line from its current position to X,Y changing all the pixels that it passes over to the current graphics foreground colour.

The three operations described above are known as 'absolute' operations. Absolute operations interpret the x,y co-ordinates given in the most obvious way – as the co-ordinates of a point on the screen. In addition to the three absolute commands there are also three 'relative' commands – MOVER, PLOTR and DRAWR. The relative commands use x and y as displacements from the graphics cursor's current position. For example,

```
PLOT X,Y
```

is an absolute command that will set the pixel at X,Y to the current graphics foreground colour but

PLOTR X,Y

is a relative command and if the graphics cursor is at xg,yg this will set the pixel at xg+X,yg+Y to the current graphics foreground colour. You can think of relative commands as specifying how far away from the graphics cursors's current position the new point is. In practice both absolute and relative commands are equally useful so it is worth being familiar with both.

As an example of using the absolute and relative graphics commands together, consider the problem of writing a subroutine that will draw a square of side H with its top left hand corner at a given position X,Y. The first thing to do is to move the graphics cursor to X,Y using an absolute MOVE command, then drawing each of the four sides can be most easily accomplished using relative DRAWR commands. That is

```
1000 MOVE X,Y
1010 DRAWR H,0
1020 DRAWR 0,-H
1030 DRAWR -H,0
1040 DRAWR 0,H
1050 RETURN
```

To see the true value of relative commands you should try to write the above subroutine using only absolute commands.

It is worth noting that if you use co-ordinate values that are off the screen then, rather than generate an error message, the Amstrad does what you ask – it carries out the MOVE, PLOT or DRAW command but off the screen where you cannot see it. Perhaps more surprisingly, if part of a DRAW command is on the screen and part off then the part on the screen will be completed successfully. In this sense the graphics screen is a sort of window onto a virtually unlimited screen.

Controlling high res colour

The ideas of logical and physical colours apply just as much to high resolution graphics as to character graphics – that is the INK command is still used to associate the logical colours available in any given mode with the actual physical colours that will be displayed. Another similarity is that the graphics commands also work in terms of a foreground and background colour but these are independent of the foreground and background colours used in character graphics and set by PEN and PAPER. The graphics background colour is set by the

```
CLG L
```

command which clears the graphics screen to logical colour L. Following this a simple CLG command, i.e. one that doesn't specify a colour, will clear the screen to logical colour L. This should be compared to the command

```
CLS
```

which will clear the text screen to the current background colour as set by the last PAPER command.

There isn't a special command to set the graphics foreground colour. Instead it can be specified as part of the PLOT, PLOTR, DRAW and DRAWR commands. For example

```
MOVE 0,0: DRAW 100,100,L
```

draws a line from 0,0 to 100,100 in logical colour L. Following this any PLOT, PLOTR, DRAW or DRAW command that doesn't specify a new graphics foreground colour will use colour L. In other words, specifying a graphics foreground colour in any graphics command sets it as the default foreground colour for all subsequent commands.

The control codes

The ASCII codes from 0 to 31 are 'control codes' in the sense that, instead of producing shapes on the screen, they act as commands. For example ASCII code 12 will clear the text screen. That is

```
PRINT CHR$(12);
```

has the same effect as

```
CLS
```

A complete list of the control codes can be seen in Chapter Nine of the 'User Instructions'. Some of the control codes have to be followed by a number of parameter codes to indicate exactly what should happen. For example, control code 14 has the same effect as the PAPER command and has to be followed by another code in the range 0 to 15 to indicate which logical colour should be assigned to the background. That is

PRINT CHR$(14);CHR$(L);

has the same effect as

PAPER L

Not all of the control codes are equivalent to BASIC commands and these are important because there is no other way of achieving the same effect. Two of these codes of particular interest, CHR$(22) transparent paper and CHR$(23) ink mode, are described in a later section.

It is worth noticing that it doesn't matter how the control codes are generated and it is not necessary to group codes in a single PRINT statement. For example, to define a graphics character in the form of a solid block you can use either

```
10 SYMBOL 240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
```

or

```
10 PRINT CHR$(25);CHR$(240);
20 FOR I=1 TO 8
30  PRINT CHR$(&FF);
40 NEXT I
```

All that matters is the order in which the codes are printed.

Using the ink mode

Control code 23 can be used to set the way that graphics foreground colour is applied to the screen. Normally if a high resolution graphics command such as PLOT is used to set a pixel to a given colour then that is exactly what happens – the pixel is set to that colour no matter what colour it was before the command. By changing the graphics ink mode, however, it is possible to make the colour to which a pixel is set depend on the colour it already is and the colour you are trying to set it to. This is a rather complicated idea but it does open up a number of powerful graphics techniques so it is worth studying more carefully. The graphics ink mode is set by

PRINT CHR$(23);CHR$(m);

where m is a parameter that determines exactly how the Amstrad affects pixels on the screen during a high resolution command. If m is 0 then the standard or normal graphics mode is put into operation. This simply means that PLOT will change the pixels it affects to the current foreground colour. Apart from this the rest of the modes are difficult to understand fully without a little knowledge of binary numbers.

For m equal to 1, 2 or 3 the colour that a pixel will be set to as the result of a high resolution command depends on its initial logical colour. The easiest way to explain how a pixel's current logical colour and the current graphics foreground colour are combined is in terms of the logical operators XOR, AND and OR. The table summarises them.

Mode	*Action*
1	new pixel colour=old pixel colour XOR current foreground colour
2	new pixel colour=old pixel colour AND current foreground colour
3	new pixel colour=old pixel colour OR current foreground colour

If you draw a line in a given colour then at the moment the only way you have of removing the line is to draw it again in the current background colour. This method works as long as the line does not pass over any areas that contain other foreground points. However if you draw a line using the XOR ink mode then drawing it a second time in exactly the same place will restore all the pixels to their original colours. The use of the graphics ink modes is described in more detail in later chapters.

Transparent paper
In the same way that the graphics ink mode can be used to control the way foreground colour is applied to the screen, the transparent mode can be used to control the way background colour is applied. Control code 22 followed by a 0 sets the normal paper mode. In this mode any pixels that are supposed to be background are set to the current background colour. However, control code 22 followed by a 1 sets the transparent paper mode and pixels that should be set to the current background colour by an operation are left unaltered. In the Amstrad 464 the only graphics command that sets pixels to the current background colour is the PRINT statement – i.e. each character that is printed

is composed of pixels that are either background or foreground pixels. Thus in normal mode printing a character changes all of the pixels within an 8 by 8 character block and each pixel is either set to the current foreground colour or the current background colour. In transprent mode, however, pixels are only changed to the current foreground colour. Any that might have been set to the background colour are left unaltered. This means that printing characters in transparent mode 'adds' them to the screen. Anything already present is not obliterated by being set to the current background colour. For example, try

```
10 MODE 1
20 PRINT CHR$(22);CHR$(1);
30 LOCATE 10,10
30 FOR I=32 TO 255
40  LOCATE 10,10
50  PRINT CHR$(I);
60 NEXT I
```

This program prints the entire character set in transparent mode at the same place on the screen and the result is that all of the pixels are eventually set to the foreground colour. If you would like to see this program in normal mode then change line 20 to

20 PRINT CHR$(22);CHR$(0);

Setting a text window
It is often the case that an application would be made easier if it was possible to divide the screen into a number of independent areas. This is indeed possible using the command

WINDOW #S,L,R,T,B

where L,R,T and B give the position and size of the new text screen or 'text window' as it is called. L is the position of the left hand edge, R the position of the right hand edge, T the position of the top and B the position of the bottom. The value of S is used to give the text window a code number that can be used by commands, such as LOCATE and PRINT, to indicate which window they refer to. For example LOCATE #1,1,1 will locate the text cursor in window number 1 to the top left hand corner. Once a text window is established it can be used in exactly the same way as the full text screen, subject to the limitations of its size and shape. For example, you can clear a text window using CLS #S, without clearing any other text windows. LOCATE #S,X,Y works in a similar way, measuring its position from the top left hand corner of text window S. In fact a text window is just like a small version of the entire screen.

As an example of how multiple text windows can be used try the following program which sets up two text windows and alternatively prints the numbers 1 to 100 on each.

```
10 MODE 2
20 WINDOW #1,10,20,1,25
30 WINDOW #2,30,40,1,25
40 FOR I=1 TO 100
50  PRINT #1,I
60 NEXT I
70 FOR I=1 TO 100
80  PRINT #2,I
90 NEXT I
```

Defining a graphics screen – ORIGIN

In the same way that you can define a number of independent text screens you can also define a graphics screen. This is defined using

ORIGIN X,Y,L,R,T,B

where L,R,T and B define the position of the graphics screen in terms of the usual graphics co-ordinate system (L, R, T and B define the left, right, top and bottom edges of the screen respectively). The first two parameters, X and Y define the position of the point 0,0 or the origin within the new graphics window. Notice that the co-ordinate system within a graphics window isn't scaled to take account of the size of the window. That is, if you define a graphics window

ORIGIN 0,0,100,200,200,100

then the bottom left hand corner of the window is 100,100 and the top right hand corner is 200,200. However if you shift the origin using

ORIGIN 100,100,100,200,200,100

then the bottom left hand corner is 0,0 and the top right hand corner is 100,100. Apart from moving the origin, the only real effect of defining a graphics window is to limit the area of the screen that the high resolution graphics commands effect. So for example, if you have one half of the screen defined as a graphics screen and the other half defined as a text screen then a CLG command will clear the graphics half to the current graphics background colour and a CLS command will clear the text half to the current text background colour. Of course if a text screen is defined inside the graphics screen then clearing the graphics screen will also clear the text screen. Similarly, clearing a text screen that contains the graphics screen will also clear the graphics screen. You can think of a text screen as an area of the display where the text commands control what happens and the graphics screen as an area where the graphics commands control what happens. An area where they overlap will be affected by both sorts of command. The command

```
ORIGIN X,Y
```

can be used to move the location of the origin to any other position without defining a new graphics window. For example, following

```
ORIGIN 320,200
```

the point 0,0 is in the middle of the screen. Moving the centre of the co-ordinate system like this doesn't affect anything already drawn on the screen but it does move the graphics cursor to the new origin.

Finding out what's on the screen – TEST and TESTR
The function TEST can be used to find out what colour pixel is on the screen at any given high resolution position. The general form of the TEST function is

```
TEST(X,Y)
```

where X,Y is the point concerned. There is also a relative form of the TEST function

```
TESTR(X,Y)
```

where xg+X,yg+Y is the point concerned (xg,yg is the position of the graphics cursor). The value returned by TEST or TESTR is the logical colour of the pixel specified or the current graphics background colour if the co-ordinates are off the screen.

Text as graphics – TAG and TAGOFF
So far the emphasis has been on how separate the text and graphics screens are but it is often the case that a display

produced with user-defined graphics would benefit from a high resolution line drawn on it. As long as the text and graphics screens overlap then this is quite possible. However, there is the problem of reconciling the two co-ordinate systems. For example, how do you draw a high resolution line between two text characters? The problem is made worse by the fact that the text co-ordinate system changes according to what mode you are using! However, you can work out the relationship between the two co-ordinate systems simply by examining the text co-ordinates of a character printed in the bottom left hand corner of the screen. The bottom left hand corner of the character is at graphics position 0,0 and the character is at LOCATE 1,25. The bottom left hand corner of a character one position up, i.e. at LOCATE 1,24 is at graphics position 0,0+char_height where char_height is the height of a character in terms of the graphics co-ordinates. In all the graphics modes char_height is 16 units. In the same way, the graphics co-ordinate of the bottom left hand corner of a character at LOCATE 2,25 is 0+char_width,0. The only trouble is char_width varies according to the mode as indicated in the table.

mode	*char_width*
0	32
1	16
2	8

Using all this information we can write two equations that will give the graphics co-ordinate of the bottom left hand corner of a character at LOCATE X,Y

graphics X=(X-1)*char_width
graphics y=400–16*Y

If you want to find the graphics co-ordinates of a point other than the bottom left hand corner within a text character then simply add the appropriate fraction of char_width and char_height to the x and y graphics co-ordinates respectively. For example, to find the middle of a character add char_width/2 to the x co-ordinate and char_height/2 to the y co-ordinate.

Using the high resolution commands to add details to a low resolution or text display is the most common way of combining high and low resolution graphics. However, it is sometimes useful to be able to PRINT a user-defined character at a location on the graphics screen. This can be achieved, but doing so may bring with it a little more than you really wanted. Following

```
TAG
```

any difference between the text and graphics screens vanishes. To be more exact, the graphics screen is the only screen available. Printing characters on the screen is controlled by the rules of the graphics screen. For example, the current printing position is controlled by the graphics cursor which is moved on by eight horizontal pixels after each character is printed so that lines of text may be formed. Also the foreground and background colours are controlled by the current graphics foreground and background colour rather than PEN and INK. Even the graphics ink mode now affects the way that characters are printed. Indeed the best way to sum up the situation following a TAG command is to say that the text screen operates according to the rule of the graphics screen.

The only other thing to be aware of when printing chracters on the graphics screen is that the graphics cursor sets the position of the top left hand corner of the character. So for example, following TAG the short program.

```
MOVE 0,0:PRINT "A"
```

will result in printing the A just off the bottom of the screen. The command

```
TAGOFF
```

restores the printing of text at the position of the text cursor.

Graphics on the 664

If you want to maintain compatability with the Amstrad 464 then you should only use the commands described above. However the Amstrad 664 does have a number of extra graphics commands and slightly more powerful versions of the familiar ones. The main change is in the handling of the graphics foreground and background colours, the graphics ink mode and the transparent paper mode. In the 664 all of the high resolution graphics commands are of the form

```
command X,Y,L,M
```

where command is one of MOVE, MOVER, PLOT, PLOTR, DRAW or DRAWR. X and Y specify the graphics position in the usual way, L is the logical colour to be used as the foreground colour and M is the graphics ink mode. (Both L and M are optional parameters.) This means that unlike the 464 the MOVE and MOVER commands can be used to set the graphics foreground colour and there is no need to use control codes to set the graphics ink mode. The control codes still work but instead of writing

```
PRINT CHR$(23);CHR$(M);
```

you can now write

```
MOVE X,Y,L,M
```

(where MOVE could be any of the high resolution commands).

The 664 also has a pair of commands that allow the graphics foreground and background colours to be set explicitly. The commands

```
GRAPHICS PEN L,T
```

and

```
GRAPHICS PAPER L
```

will set the graphics foreground and background colours respectively and set the paper transparency to T.

Other graphics commands that are present in the 664's version of BASIC and their functions are as follows:

MASK P,F	Sets the pattern of dots that make up a line to P, i.e. it enables the drawing of dotted lines. If F is 1 then the endpoints of a line are drawn. If it is 0 then the endpoints are not drawn.
FRAME	Waits for the start of the next TV frame. 464 users get the same effect by CALL &BD19.
FILL L	Fills an area bounded by the current graphics foreground colour with colour L starting from the current graphics cursor position.

COPYCHR$	A function that returns a string containing the character under the current position of the text cursor. If the character cannot be recognised then a null string is returned.

Some of these commands are described, along with the way that their effects are obtained on the 464, in later chapters.

Conclusion

The Amstrad's graphics system may sometimes seem a little complicated. However, the system is very logical and if you concentrate on seeing the way in which it works rather than memorising commands you should find it easy. In the rest of this book the emphasis falls on graphics methods, rather than tricky ways of using the Amstrad, but there are a great many novel ways of using the graphics commands introduced along the way. If at any time you forget what a command does, then you will find the Graphics Commands table in Appendix I a useful resume.

2 Animation in text mode using sprites

Animation is one of the most enjoyable areas of computer graphics. The most obvious application is in the production of action computer games, but it also turns up in many other more serious programs. So, even if you are not interested in computer games it is worth knowing how animation works. The basic method of animation is simple, but if it is applied without care the result can be a messy program that produces very poor movement. In this chapter the idea of a 'sprite' is introduced as a way of organising the movement of a number of objects on the screen. To make things easier at first only character graphics will be used.

Blanking animation

The standard method of making things move in a computer display is fairly well known even to novice programmers. To move a shape on the screen you first draw it at one location, then remove it or 'blank' it out, and then redraw it slightly shifted. The smoothness of the apparent motion produced by this blanking animation depends on a number of interacting factors. The following guidelines should aid the production of smooth movement:

1 The distance moved at each step should be small.
2 The time taken to blank and produce the shape should also be small.
3 The shape should be displayed on the screen for as long as possible between moves.
4 For really smooth motion each step should be synchronised with the TV frame display rate.
5 The perceived smoothness is also affected by the shape, colour and texture of the object.

The trouble is that, in practice, most micros are too slow to produce smooth animation using BASIC. Even using assembler, a great deal of care and attention is needed to produce a flicker-free screen. The problem increases as the number of separate objects being moved increases. Normally animation is developed in a haphazard way, with each moving object being added to the program in turn, perhaps even using a completely different method for each. However, there is a way of organising the animation of any number of small shapes that is entirely systematic.

Sprites

Once you start writing programs to produce animation you soon learn that the best way to think of the path that a moving object takes is by using the idea of velocities. Velocity can be thought of as 'speed in a given direction' and so it summarises two elements of movement, rate of travel and direction of travel. So, for example, if an object is moving smoothly in a straight line across the screen it can be associated with four quantities

X – its present x co-ordinate
Y – its present y co-ordinate
XV – its x or horizontal velocity
YV – its y or vertical velocity

In other words, at any given point in the program the object is at the position X,Y and its next position will be X+XV,Y+YV. The use of velocities makes it particularly easy to update the current position to give the new position using nothing but addition – which is one of the fastest operations that any computer can perform. In practice, moving an object about the screen rarely involves nothing but motion in a straight line! Fortunately it is not difficult to extend the use of velocities to produce all types of movement.

A small graphics shape together with its current position and a pair of velocities is usually called a 'sprite'. (They are also known as 'MOBs' standing for 'Movable OBjects'.) This sort of sprite is the simplest used in animated graphics and in practice the idea is elaborated considerably. Sprites are often associated with special graphics hardware that will automatically update positions and plot the shape on the screen. However, while sprite hardware does make things easier and faster, software sprites are also useful in writing programs on even the least sophisticated computer.

Acceleration

Before giving an example of how sprites can be used it is necessary to introduce the idea of acceleration. As already noted, it is rare that objects move smoothly in straight lines. For one thing they tend to go off the edge of the screen! In practice objects change their velocity as they move across the screen, sometimes continuously as in the case of a ball or a lunar lander 'falling' in an arc down the screen, and sometimes suddenly as in the case of an object 'bouncing' off the sides of the screen. However, the common factor is that the velocity does change and a change in velocity is usually called an 'acceleration'. This suggests that to make sprites really useful the number of quantities associated with a sprite should be increased to six by adding a horizontal and a vertical acceleration, XA and YA. Now at each stage the current position is updated by

X=X+XV
and
Y=Y+YV
and the velocities by
XV=XV+XA
and
YV=YV+YA

This method does indeed work for movements that involve smooth changes in an object's velocity, but what about the sudden changes involved in an object 'bouncing' off another object? To take this sort of sudden change into account we have to introduce yet another idea – that of a 'force function'. A force function defines the way that acceleration depends on current position, current velocity, current acceleration and any other conditions that you might want to take into account! It is easier to understand the way that a force function controls the movement of a sprite by looking at an example. But first it is worth summarising the definition of a simple sprite. A Sprite is composed of the following information

1 The shape of the object.
2 The current position stored in X and Y.
3 The velocities XV and YV.
4 The accelerations XA and YA.
5 The force function that is used to update accelerations.

A simple sprite example

To illustrate the ideas introduced so far the simple and well known problem of bouncing a ball around the screen is a good starting point. However to make the problem a little more obvious, consider the problem of bouncing more than one ball at a time. In fact the program given below will bounce any number of balls around the screen if you don't worry about how fast (or rather how slowly) everything moves!

The first part of constructing the program involves defining a suitable sprite for the bouncing ball. The shape of the ball is easily solved by using graphics character given by CHR$(231) stored in a string S$. In other words, PRINT S$ produces the ball shape on the screen. Its current position can be stored in X and Y and its current velocity in XV and YV. The problems really only start when you try to work out what the acceleration should be. When the ball is moving about the screen away from the wall the answer is easy – both horizontal and vertical acceleration are 0. However, when the ball is near, or to be more accurate touching, the edge of the screen the acceleration is clearly not zero because the one of the ball's velocities will be reversed to create the bounce. If the ball meets a horizontal wall then YV is changed to −YV, if it meets vertical wall then XV is changed to −XV. In terms of acceleration this implies

```
IF the ball is in contact with a horizontal wall THEN
YA=−2*YV ELSE YA=0
IF the ball is in contact with a vertical wall THEN
XA=−2*XV ELSE XA=0
```

To turn these two IF statements completely into BASIC requires only that the test for horizontal and vertical walls is made exact. If the vertical walls are placed at X=1 and X=40 and the horizontal walls at 1 and 25 (i.e. at the edges of a mode 1 screen) then the tests can be written

```
IF Y=1 OR Y=25 THEN YA=−2*YV ELSE YA=0
IF X=1 OR X=40 THEN XA=2*XV ELSE XA=0
```

You should recognise these two IF statements as the force function for the ball because together they determine the acceleration at each update. In most cases however there is one addition that has to be made to IF statements that test for a collision with a wall or any other object on the screen. The problem is that during one of the position updates the sprite could have moved 'through' the wall and the test should take this into account. Also if the sprite has moved through a wall it really ought to be moved back to the correct side. If these considerations are taken into account the tests become

```
YA=0
XA=0
IF Y<=1 THEN Y=1:YA=-2*YV
IF Y>=25 THEN Y=25:YA=-2*YV
IF X<=1 THEN X=1:XA=-2*XV
IF X>=40 THEN X=40:XA=-2*XV
```

(Notice how the ELSE part of the IF statements has had to be replaced by the default setting of YA and XA before the tests are carried out.)

Now all that remains is to put all of the parts of the sprite together to produce the program. However, the whole point about using sprites is that once you have defined a sprite for an object it is easy to re-use the definition to produce any number of examples of the sprite moving around the screen. All that you have to do is to replace each of the variables associated with the sprite by an array large enough to hold the information for each sprite. For example, if you want five balls bouncing round the screen then use X(1) to X(5) to record the current x co-ordinate of each sprite. X(1) is the x co-ordinate of the first example of the sprite, X(2) the x co-ordinate of the second and so on. The resulting program is surprisingly easy

```
10    REM sprite animator
20    MODE 1:BORDER 26
30    PRINT "HOW MANY SPRITES ";
40    INPUT N%
50    CLS
60    GOSUB 1000
70    WHILE TIME-RTIME<10000
80     GOSUB 2000
90     GOSUB 3000
100    T%=T%+1
110   WEND
120   END

1000 REM init
1010 DIM S$(N%),X(N%),Y(N%),XV(N%),YV(N%)
1020 DIM XA(N%),YA(N%),PX(N%),PY(N%)
1030 FOR I%=1 TO N%
1040  S$(I%)=CHR$(231)
1050  X(I%)=INT(RND*37+2)
1060  Y(I%)=INT(RND*22+2)
1070  PX(I%)=X(I%)
1080  PY(I%)=Y(I%)
1090  XV(I%)=SGN(RND-0.5)
1100  YV(I%)=SGN(RND-0.5)
```

```
1110  XA(I%)=0
1120  YA(I%)=0
1130 NEXT I%
1140 RTIME=TIME
1150 T%=0
1199 RETURN

2000 REM animate
2010 FOR I%=1 TO N%
2020  PX(I%)=X(I%)
2030  PY(I%)=Y(I%)
2040  X(I%)=X(I%)+XV(I%)
2050  Y(I%)=Y(I%)+YV(I%)
2060  XA(I%)=0
2070  YA(I%)=0
2080  IF X(I%)<=1 THEN X(I%)=1:XA(I%)=-2*XV(I%)
2090  IF X(I%)>=40 THEN X(I%)=40:XA(I%)=-2*XV(I%)
2100  IF Y(I%)<=1 THEN Y(I%)=1:YA(I%)=-2*YV(I%)
2110  IF Y(I%)>=25 THEN Y(I%)=25:YA(I%)=-2*YV(I%)
2120  XV(I%)=XV(I%)+XA(I%)
2130  YV(I%)=YV(I%)+YA(I%)
2140 NEXT I%
2999 RETURN

3000 FOR I%=1 TO N%
3010  LOCATE PX(I%),PY(I%)
3020  PRINT SPC(1);
3030  LOCATE X(I%),Y(I%)
3040  PRINT S$(I%);
3050 NEXT I%
3999 RETURN
```

Subroutine 1000 simply initialises all of the arrays, X(I) and Y(I) are sprite I's position, XV(I) and YV(I) are its velocities and XA(I) and YA(I) are its accelerations. The arrays PX and PY are used to hold each sprite's previous position. The sprites all start off from random positions and velocities. Subroutines 2000 and 3000 perform the animation, 2000 updates all of the sprite values and 3000 blanks and reprints each sprite in turn.

The power of the sprite method of organising animation programs should be clear from the economy and simplicity of the above program. However, the sprite method also leads to programs that are easy to modify. For example, if you want to add some 'gravity' to the bouncing balls so that they fall in a parabolic arc and then bounce back up to the same height change line 2070 to

```
2070 YA=0.2
```

The only change is to set the vertical acceleration to a small constant to mimic the effect of gravity.

Other changes are equally simple. For example, if you want to make the balls bounce less each time change lines 2100 and 2110 to

```
2100 IF Y(I%)<=1  THEN Y(I%)=1 :YA(I%)=-1.99*YV(I%)
2110 IF Y(I%)>=40 THEN Y(I%)=40:YA(I%)=-1.99*YV(I%)
```

where the −1.99*YV reflects a reduction in the vertical velocity on each (horizontal) bounce. You can make each of the sprites have a different shape by adding

```
1040 S$(I%)=CHR$(231+I%)
```

You could even attach a colour code, a sound to be produced at each bounce, etc to the sprites – but more of this later.

Priorities, collisions and events

You may be thinking that bouncing balls around the screen, no matter how many, hardly demonstrates that sprites are a vital tool. However, once you start exploring some of the additional ideas that suggest themselves and seem natural when you work with sprites, it becomes difficult to believe that you ever thought about animation in any other way! For example, the order in which you blank and draw sprites imposes a natural priority on a collection of sprites. A later sprite will appear to pass in front of an earlier sprite if they both happen to reach the same screen position. This is simply because the later sprite will overwrite the earlier one but it leads to a way of ordering sprites so that when they meet you can predict what the result will look like. The idea of one sprite passing over another one leads naturally on to the idea of sprite-sprite collisions. At the end of all sprite moves it is easy (but time consuming in BASIC) to check to see if any of the sprites are in the same position and so have collided. A sprite collision is usually dealt with by a special subroutine. For example, if one of the sprites is a rocket and the other a target then, when a sprite-sprite collision is detected, the obvious thing is to call a subroutine that produces an explosion!

This idea of a collision causing something different to happen within the program is similar to what happens during a hardware interrupt. Interrupts are normally used to inform the computer that something unusual or important that needs special attention has happened in the outside world. This is exactly what happens with a sprite collision, except that the event is internal. The idea can be generalised to sprite events which correspond to any detectable condition that should cause the program to do something different. For example, the sprite bounce from the boundary wall that was controlled by the force function could have been declared a sprite event that, when detected, called a subroutine that reversed the velocity and perhaps made a noise. There is often more than one way to animate a sprite and the number of ways increases as the idea of a sprite becomes more and more elaborate. Some of these ideas are described later but there is a lot to be gained from trying to keep sprites simple – especially in BASIC!

The animation loop

So far the question of the best way to implement sprites has not been discussed. It may come as something of a surprise to discover that there are a number of different ways of implementing the sprite idea and they differ in terms of their simplicity and their efficiency.

All animation is done in terms of an 'animation loop'. The only real question is how much should be done each time through the loop. In the ideal animation loop all of the sprites' variables would be updated, then all of the sprites would be blanked and reprinted at their new positions. In this case a loop counter or animation counter, T say, would represent the number of times all of the sprites had moved. In many applications the value in T can be treated as a measure of the time that the program has been running. Before this loop is repeated again a routine is used to check for any sprite events – collisions etc – that might have occurred. If an event is detected then a special part of the program is called that handles the event. What happens after this handling depends very much on its nature. Some events signal the end of the program and others cause the animation loop to be restarted.

The only trouble with this ideal animation loop is that it assumes that the time to do the sprite updates is short. In fact if this time is much longer than one or two TV frame periods the user begins to see the sprites moving in a stop-start, jerky way. Of course if you are using BASIC then this is almost certain to be the case and a modification to this fast animation method is called for. In the 'sprite at a time' animation loop only one of the sprites is moved each time through the animation loop. Thus if there are N sprites it takes N times through the loop before they are all updated. In this case the loop counter no longer records the total number of times all of the sprites have moved and it cannot be interpreted as a simple time variable. Notice that it is important to check for any sprite events after all of the sprites have been updated. The reason for this is that it is essential to check for sprite events on the basis of the screen as it appears to the user. For example, if you checked for a collision between two sprites during the animation loop, the chances are that you would incorrectly detect a collision between the new position of one sprite and the old position of the other! If you would like to see the effect of 'a sprite at a time' animation make the following changes to the original sprite program

```
90 REM

2140 LOCATE PX(I%),PY(I%)
2150 PRINT SPC(1);          14
2160 LOCATE X(I%),Y(I%)
2170 PRINT S$(I%);
2180 NEXT I%
```

The ideal animation loop is the better of the two methods because the user never sees the screen in an intermediate state, with some of the sprites in their old positions and some in their new positions. However if you are using BASIC or animating a great many sprites then the 'sprite at a time' method is the only way of producing smooth animation.

A classic example – the space invader sprite

You might be thinking that some of the examples of moving graphics that you have seen would be difficult to implement using sprites. This is not the case. None of the animated displays I have seen is difficult to interpret in terms of sprite graphics, with a little practice. For example, consider the traditional space invaders screen with alien ships in rows moving from side to side and then slowly moving down the screen. How can this almost static movment be translated into sprites? The answer is surprisingly easy! Change the following lines in the original bouncing balls program

```
1050 X(I%)=I*3
1060 Y(I%)=2
1090 XV(I%)=2
1100 YV(I%)=.1

2070 YA(I%)=0
2080 XA(I%)=-2*XV(I%)
2090 REM
2060 XA(I)=-2*XV(I)
```

The only changes are to the initial positions and velocities of the shapes and to the force formula. The initital positions are set to produce a single row of shapes by line 1050 and 1060. Lines 1090 and 1100 set a horizontal velocity of 2 and a vertical velocity of .1. As you might expect, this results in a large sideways velocity but a very small vertical velocity making the row creep forward. Line 2080 is responsible for the side to side oscillation because it reverses the velocity at each update. You should now be able to see how this method could be extended to more than one row and how to make the 'ships' move faster down the screen as the game progresses without having to use a single non-sprite idea!

Practical animation – integer variables
In practice the main concern of any animation program is speed. The faster you can make the animation loop run the more sprites you can animate. To this end, it is standard practice to use integer variables, rather than real variables, wherever possible. Integer variables are certainly capable of holding the co-ordinate range of both the high and low resolution screens and they are always faster, so you may be wondering why they cannot always be used in preference to the slower real variables. The answer is that for most sprite applications at least one of the positions, velocity or acceleration variables is fractional, and hence cannot be stored in integer variables. This need to use real variables can present something of a problem if you attempt to gain a speed increase by using assembler. The reason is that real arithmetic is very difficult to carry out using assembler without a great deal of programming. However it is usually possible to avoid using real arithmetic by working with every value multiplied by a suitable 'scale factor' large enough to remove any fractional part that might occur.

Internal animation
One of the most impressive features of 'space invaders' on many computers is the way that each of the invaders 'waves its arms' as they move from side to side. So far all the sprites that have been described have kept their shape fixed as they moved. However, internal animation, as exemplified by space invaders, is not difficult to achieve, and what is more surprising, adds very little to the complexity or time taken to animate a sprite.

The key to internal animation is to replace the string that holds the single shape normally associated with a sprite by a string array that holds a range of shapes. The shapes stored in the array form a sequence that will produce the desired internal animation. As the sprite moves around the screen the animation counter T is used to generate an index that governs which of the shapes will be printed on the screen by the animation loop. Perhaps the easiest way to explain this idea is by way of an example.

The following program will animate a little man-shaped figure as it falls toward the bottom of the screen. The internal animation takes the form of making the man tumble as he falls. The sequence of shapes necessary to implement this internal animation can be seen in fig 2.1.

Fig 2.1 Sequence of shapes needed to animate the tumbling man

The full animation sequence involves eight different stages. First shape 0 is used then shape 1 and so on to shape 7 then the whole cycle repeats itself. Thus if the shapes are stored in the array S$(7) the animation sequence is S$(0), S$(1) . . . S$(7), S$(0) and so on. Using this arrangement the animation counter can easily be used to select the correct shape during the animation. In other words if the animation counter starts from 0 we have

T=	0	1	2	3	4	5	6	7	8	9
Use	S$(0)	S$(1)	S$(2)	S$(3)	S$(4)	S$(5)	S$(6)	S$(7)	S$(0)	S$(1)

If you look at this pattern carefully you should be able to see that the index of the array is simply the remainder when you divide T by 8, that is T MOD 8. This is exactly the method used to select which shape should be displayed using the animation counter. If the internal animation sequence consists of N shapes then the index is simply the remainder after dividing the animation counter by N.

Returning to the tumbling falling man program this gives

```
10   REM internal animation
20   MODE 1
30   GOSUB 1000
40   GOSUB 2000
50   WHILE Y<24
60    T%=T%+1
70    FOR I%=1 TO 300:NEXT I%
80    GOSUB 3000
90    GOSUB 4000
100  WEND
110  END
```

```
1000 REM set up
1010 INK 0,0
1020 INK 1,26
1030 PEN 0
1040 PAPER 1
1050 CLS
1999 RETURN

2000 REM init
2010 SYMBOL 240,&18,&DB,&7E,&18,&3C,&66,&42,&42
2020 SYMBOL 241,&33,&1F,&E,&7E,&CB,&89,&18,&30
2030 SYMBOL 242,&3,&E6,&34,&1F,&1F,&34,&E6,&3
2040 SYMBOL 243,&30,&18,&89,&CB,&7E,&E,&1F,&33
2050 SYMBOL 244,&42,&42,&66,&3C,&18,&7E,&DB,&99
2060 SYMBOL 245,&C,&18,&91,&D3,&7E,&70,&F8,&CC
2070 SYMBOL 246,&C0,&67,&2C,&F8,&F8,&2C,&67,&C0
2080 SYMBOL 247,&CC,&F8,&70,&7E,&D3,&90,&18,&C
2100 DIM S$(7)
2110 FOR I%=0 TO 7
2120  S$(I%)=CHR$(240+I%)
2130 NEXT I%
2160 X=10
2170 Y=3
2180 XV=0
2190 YV=1
2200 XA=0
2210 YA=0
2990 RTIME=TIME
2999 RETURN

3000 REM update
3010 XP=X:YP=Y
3020 X=X+XV
3030 Y=Y+YV
3040 XV=XV+XA
3050 YV=YV+YA
3999 RETURN

4000 REM reprint
4010 R=T% MOD 8
4020 LOCATE XP,YP:PRINT " ";
4030 LOCATE X,Y:PRINT S$(R);
4999 RETURN
```

Subroutines 1000 and 2000 initialise the display mode and the sprite respectively. As there is only one sprite there is no need to use arrays to store position, velocity and acceleration. Subroutine 3000 performs the usual sprite updates, and subroutine 4000 blanks out and reprints the sprite. The only real difference between this program and the bouncing balls sprite program given in the previous chapter is the way that the animation counter is used to select which shape will be printed.

Notice that as the falling man only moves vertically there is really no need to carry out any updates on the X position, velocity etc. However to make the program completely general, if a little slower than it need be, these unnecessary updates are included. You can introduce internal animation to any number of sprites in exactly the same way subject, of course, to the condition that you can move everything fast enough. It is also possible to use other variables than the animation counter to index the array of different shapes. For example an animation of a walking man might use the X co-ordinate to select one of a number of different 'walking positions'. However the animation counter serves for most purposes.

Sprite events

The idea of a sprite event was introduced earlier – a sprite event is any detectable condition that requires something other than the standard animation sequence to handle it. The best way to illustrate this idea is to add a simple event detector and handler to the falling man program given in the last section. At the moment when the man finally reaches the end of his fall nothing exciting happens. To rectify this simply entails two extra subroutines.

```
95 GOSUB 5000

5000 REM detect event
5010 IF Y>22 THEN GOSUB 6000
5020 RETURN

6000 REM event
6010 SOUND 1,1000,25,15,,,31
6020 END
```

Subroutine 5000 checks if the man has reached the ground or not and transfers control to the event handler, subroutine 6000, if he has. In this case the event handler only makes an explosive noise and then stops the program but in principle it could be a much more complicated routine than this.

As a more complicated example, the following modifications to the bouncing ball multiple sprite program given earlier will add a crash noise each time a collision occurs and remove the pair of sprites involved.

```
95 GOSUB 4000

4000 REM detect event
4010 FOR I%=1 TO N%
4020  IF S$(I%)="" THEN GOTO 4070
4030  FOR J%=1 TO N%
4040   IF I%=J% THEN GOTO 4060
4050   IF X(I%)=X(J%) AND Y(I%)=Y(J%) THEN GOSUB 5000
4060  NEXT J%
4070 NEXT I%
4080 RETURN

5000 REM event handler
5010 S$(I%)=""
5020 S$(J%)=""
5030 SOUND 1,50,20
5040 RETURN
```

The above lines should be added to the first version of the bouncing ball program. The collision check routine, subroutine 4000, isn't the most efficient possible as it checks each pair of sprites twice but it does have the advantage of being easy to understand.

Explosions – terminal events!

The most common use of sprite events is to detect a condition that implies the end of a game. For example, if a missile hits its target then the sprite collision involved would signal the end of the game. It is surprising how often end of game sequences involve animated explosions. It is possible to write pages on how sprites can meet their final end in spectacular explosions but the main trouble is that most of them need the speed of assembler to be effective. The simplest and quickest method to produce an explosion is to define an explosion character that can be printed over the current sprite's position. To define a good explosion character is not difficult but you should try to avoid using too many of the pixels near the edge of the character so that the outline of the 8 by 8 character square is invisible.

More advanced methods of constructing explosions are nearly all based either on making the explosion appear to grow by painting a sequence of explosion characters or on using high resolution graphics commands to make the pixels of the shape appear to move apart. To do this smoothly and convincingly needs assembler and indeed a good explosion routine can take an unreasonable amount of code to produce! However, you might like to try either of the following two explosion routines. The first makes use of a short sequence of expanding explosion characters

```
10  MODE 0
20  SYMBOL 240,0,0,0,&01,&08,0,0,0
30  SYMBOL 241,0,0,&04,&18,&18,&14,0,0
40  SYMBOL 242,0,&04,&24,&1C,&72,&14,&24,0
50  SYMBOL 243,&82,&24,&3C,&1C,&FC,&7A,&48,&85
60  FOR I=240 TO 243
70   LOCATE 10,10
80   PRINT CHR$(I);
90   FOR J=1 TO 50:NEXT J
100 NEXT I
110 FOR J=1 TO 1000:NEXT J
120 GOTO 10
```

and the second simply plots radial lines from the centre of the character

```
10 MODE 2
20 X=320
30 Y=200
40 DX=25-INT(RND*50)
50 DY=50-INT(RND*50)
60 MOVE X,Y
70 DRAWR DX,DY
80 GOTO 40
```

Notice that to use the above routine in a low resolution graphics program involves converting low res co-ordinates to high res co-ordinates. In general explosion routines are very much improved by good sound effects and this makes sound an even more important components of BASIC explosions, to compensate for its limitations of speed.

The outside world

Interactive control of sprites is such an obvious feature of nearly every computer game that it is often dismissed as a small problem of program implementation, when in fact it is quite central to the success or otherwise of the game. The way that the user interacts with the game is important because it sets the level and type of skill required to be a successful player. A poor or inaccurate method of controlling the sprites involved in the game simply raises the frustration level of the user, because of the difficulty in getting the sprites to do what is required. If you regularly play computer games you cannot fail to have experienced the feeling that you've pressed a key or moved the joystick in time to avoid destruction but the sprite that you were controlling just didn't take any notice! On the other hand an accurate and natural method of controlling sprites produces a feeling of involvement that may raise a run of the mill game to new heights of popularity.

The methods that you can use to control sprites are to a certain extent governed by the hardware that you have available. The main method of sprite control is the keyboard and so it is important to look at the different way in which this can be used.

General sprite control

So far the sprite animation loop – blank sprites, update sprites, plot sprites – has only been used to produce sprites that move around the screen bumping into things and occasionally exploding. However, interactive animation programs must allow the user a way of controlling at least one of the sprites in the display. This causes no real problem for the theoretical framework of sprite animation that has been constructed. The obvious place to insert any user control is in the 'update sprite' part of the loop. Also the theory implies that whatever method of control is used it should be implemented as part of the force function. The simplicity of the sprite is partly due to the fact that its position and its velocity are automatically updated in the same way each time through the animation loop and any variation in motion comes from the force function. It is also this fact that makes it easier to implement sprites in hardware – the regular updating can be carried out by hardware and the simple but more varied behaviour of the force function can be left to the software. As has already been demonstrated, it is possible to exercise any sort of control over the motion of sprites using the force function but it isn't always the most direct approach. In particular, user control over sprites is rarely directly in terms of a force function and it is usually more efficient, if a little less tidy, to alter the other quantities associated with sprites. User control can be classified according to which quantity is directly affected

Position	– direct updating of the x and y co-ordinates
Velocity	– direct updating of the x and y velocities
Acceleration	– direct updating of the force function

Each of these types of control actually occurs in games and other animation programs, each one has something different to offer and, what is more surprising, each one is encountered often enough to be considered as an important standard technique. For this reason each one will be dealt with in turn.

Positional control

Updating the x and y co-ordinates directly is the simplest form of control. When a keyboard is used as the input device every possible direction of movement is associated with a particular key. Each time through the animation loop the keyboard is inspected and, according to which key is pressed, the position of the controlled sprite is updated. For example

```
10  MODE 1
20  X=20:Y=20
30  LOCATE X,Y
40  PRINT SPC(1);
50  IF INKEY(0)=0 THEN Y=Y-1
60  IF INKEY(2)=0 THEN Y=Y+1
70  IF INKEY(1)=0 THEN X=X+1
80  IF INKEY(8)=0 THEN X=X-1
90  LOCATE X,Y
100 PRINT CHR$(231);
110 GOTO 30
```

This simple program animates a ball around the screen. The INKEY function is used to test each of the cursor keys in turn (line 50 to 80) and the sprite's position is updated according to which, if any, are pressed.

The only trouble with this method of moving something around the screen is that it can take too long to move large distances (particularly when something is being moved using high resolution co-ordinates, see Chapter Three). If the amount that was added to the X and Y co-ordinates was increased it would allow the sprite to be moved large distances more quickly but it would make the exact positioning impossible. The best solution to this problem is to allow the use of two different amounts by which the sprite can be moved. This may sound complicated but it is very easy to implement if instead of updating the X and Y co-ordinates directly the IF statements set increments to be added to them. For example

```
10  MODE 1
20  X=20:Y=20
30  LOCATE X,Y
40  PRINT " ";
50  DX=0:DY=0
60  IF INKEY(0)=0 THEN DY=-1
70  IF INKEY(2)=0 THEN DY=1
80  IF INKEY(1)=0 THEN DX=1
90  IF INKEY(8)=0 THEN DX=-1
100 IF INKEY(9)=0 THEN DX=DX*4:DY=DY*4
110 X=X+DX:Y=Y+DY
120 LOCATE X,Y
130 PRINT CHR$(231);
140 GOTO 30
```

Lines 60 to 90 set DX and DY to an appropriate increment according to which of the cursor keys is pressed. A larger increment is achieved by multiplying DX and DY by 4 if the COPY key is pressed (line 100).

True positional control – the joystick

Although using the cursor keys in the way described in the previous section is a way of directly updating a sprite's X and Y co-ordinates it is not really true positional control. True positional control involves setting the sprite's X and Y co-ordinates to some specified position, not their repeated updating by adding an increment. However this sort of positional control is only possible with input devices that produce screen co-ordinates such as proportional joysticks or lightpens. With a proportional joystick or a light pen you can effectively point at a position on the screen and this supplies your program with a pair of co-ordinates where you would like the sprite to be positioned – thus updating the form of setting the sprite's X and Y variables to the co-ordinates supplied by the input device. True positional control allows the user to move a sprite from one place on the screen to another without having to move through any intermediate positions.

True positional control is often very useful, even though it brings certain problems of its own, but it is not available on a standard Amstrad. The main reason for this is that the Amstrad joysticks are not proportional joysticks but simply an extension of the main keyboard. That is they do not supply X and Y co-ordinates to the machine only an indication of UP, DOWN, LEFT, RIGHT and FIRE. In this sense, for sprite control, the joysticks are entirely equivalent to the cursor keys. For example, to change the first positioning program given in the previous section to use joystick zero simply change the INKEY parameters in lines 50, 60, 70 and 80 to

```
50 IF INKEY(73)=0 THEN Y=Y-1
60 IF INKEY(72)=0 THEN Y=Y+1
70 IF INKEY(75)=0 THEN X=X+1
80 IF INKEY(74)=0 THEN X=X-1
```

Velocity control

Velocity control is not used very much with joysticks because positional control seems to suit the situation much better. However, velocity control has a lot to offer if the only input device is a keyboard. Although this form of control is correctly known as velocity control, from the user's point of view it often seems more like directional control. The fundamental idea is that each time through the animation loop the keyboard is inspected and depending on which key, if any, is pressed the sprite's velocity is updated. In most cases a change in velocity brings about a change in the direction of motion rather than in the speed of the sprite – although a change in speed is sometimes useful. As an example of velocity control, consider the animation of a 'bat', first using positional control

```
10 MODE 1
20 X=20
30 Y=20
40 IF INKEY(8)=0 THEN X=X-1
50 IF INKEY(1)=0 THEN X=X+1
60 LOCATE X,Y
70 PRINT SPC(1);STRING$(3,143);SPC(1);
80 GOTO 40
```

This program moves the bat one position per keypress and as such is a little slow and tedious. (If you are wondering why there is no PRINT statement to erase the bat from the screen it is because the two spaces one at each end of the bat do the job automatically – i.e. the sprite is 'self blanking'. Self blanking is duscussed more fully in the next chapter.) Now try the velocity control approach

```
10  MODE 1
20  X=20
30  Y=20
40  VX=1
50  X=X+VX
60  IF X<1 THEN X=1
70  IF X>35 THEN X=35
80  IF INKEY(8)=0 THEN VX=-1
90  IF INKEY(1)=0 THEN VX=1
100 LOCATE X,Y
110 PRINT SPC(1);STRING$(3,143);SPC(1);
120 GOTO 50
```

You will discover that the bat drifts either right or left, depending on which of the left or right cursor keys you press. Notice that the position of the bat is updated automatically each time through the animation loop even is no key is pressed and that it is the velocity VX which is changed in response to a keypress. This example is typical of velocity control in that things keep on moving even if you don't do anything. However, this simple one-dimensional (side to side) movement and control is not the only way that velocity control can be used. For example, as well as the obvious generalisation to two dimensions using four keys to control the up/down, left/right velocity of a sprite, you can introduce a single or double key control that 'turns' the sprite. For example

```
10  MODE 1
20  DATA 0,-1,1,0,0,1,-1,0
30  DIM V(4),W(4)
40  FOR K=1 TO 4
50   READ V(K),W(K)
60  NEXT K
70  X=20
80  Y=12
90  K=1
100 LOCATE X,Y
110 PRINT SPC(1);
120 X=X+V(K)
130 Y=Y+W(K)
140 IF X=1 OR X=40 THEN K=K+2
150 IF Y=1 OR Y=25 THEN K=K+2
160 IF INKEY(9)=0 THEN K=K+1
170 IF K>4 THEN K=K-4
180 LOCATE X,Y
190 PRINT "*";
200 FOR J=1 TO 50:NEXT J
210 GOTO 100
```

This program uses two arrays V and W to hold pairs of x and y velocities as shown in fig 2.2. The direction in which the asterisk is travelling is determined by the current value of K. Each time the copy key is pressed the value of K is incremented by 1 and a new pair of velocities is selected. This action makes the sprite turn through 90 degrees with each press of the space bar. By adding four more elements to the arrays you can make a sprite turn through 45 degrees. This type of control is particularly effective if the shape of the sprite is changed so as to point in the direction of motion. The method of doing this exactly the same as described for internal animation only instead of using the animation counter as an index to select the shape, the direction index (K in the above example) is used. To see this add the following lines to the above program

```
35  DIM S(4)

65  S(1)=240:S(2)=243:S(3)=241:S(4)=242

190 PRINT CHR$(S(K));
```

Acceleration control

Acceleration control is not quite as common as the previous two

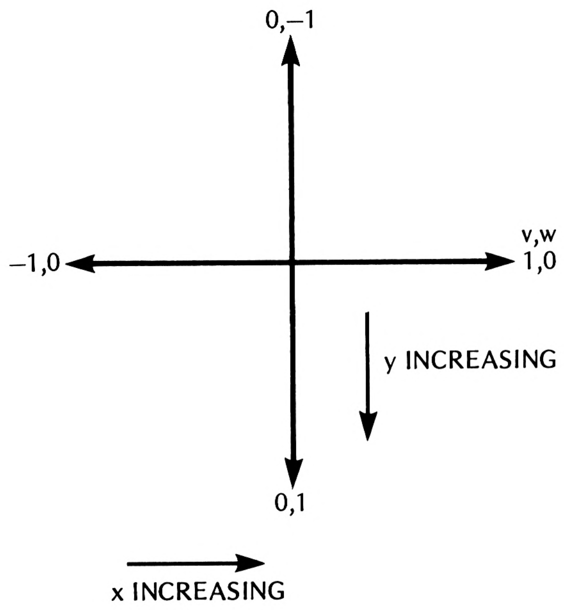


Fig 2.2 The directions of movement produced by the values of v and w

methods but it does have one very important use. In games based on simulations of landing or flying spaceships, firing a rocket produces a change in the velocity of the sprite by applying a constant acceleration as long as a key is pressed. This sort of control is most simply implemented by directly updating the sprite's acceleration variables each time through the animation loop. For example, if a sprite is falling down the screen because of a constant vertical acceleration simulating gravity then a 'landing' game can easily be produced

```
10  MODE 1
20  X=20
30  Y=1
40  VY=0
50  AY=0.001
60  LOCATE X,Y
70  PRINT SPC(1)
80  Y=Y+VY
90  VY=VY+AY
100 IF INKEY(9)=0 THEN AY=-0.001 ELSE AY=0.001
110 LOCATE X,Y
120 PRINT CHR$(239);
130 FOR J=1 TO 20:NEXT J
140 GOTO 60
```

The 'rocket' engine can be 'fired' by pressing the COPY key and this will change the acceleration from ·001 to −·001, so slowing the descent. All that is necessary to change this into a full 'lander' game is the addition of a fuel count and a sprite event detector for 'touch down'.

A free fall game

The following game shows most of the ideas introduced in this chapter working together. The program uses the falling man sprite given earlier, with the addition of an aeroplane and a parachute to create a simple 'free fall' game. At the start an aeroplane flies across the top of the screen and a target appears at the bottom. By pressing the down arrow key the man can be made to jump from the plane and free fall, hopefully toward the target. There is a constant wind blowing so this is not as easy as it sounds! By pressing the COPY key a parachute can be made to open and from this point on the man can be steered to the right or left using the cursor keys.

The structure of the program is fairly straightforward and you should be able to make sense of it with the aid of the subroutine table given below.

Subroutine	*action*
1000	set up colours and target position
2000	define shapes and set up shape strings
2500	initialise sprite variables for plane
2700	initialise sprite variables for falling man
2900	initialise sprite variables for parachute
3000	update sprite variables
4000	print sprite as falling man
5000	print sprite as plane
6000	test left and right cursor keys and set velocity
7000	print sprite as parachuting man
8000	end game routine
9000	pause
9500	clear keyboard buffer

The only subroutine whose purpose isn't clear is 9500. The Amstrad has a keyboard buffer that is capable of storing a number of keypresses. To avoid these being accepted as the answer to the

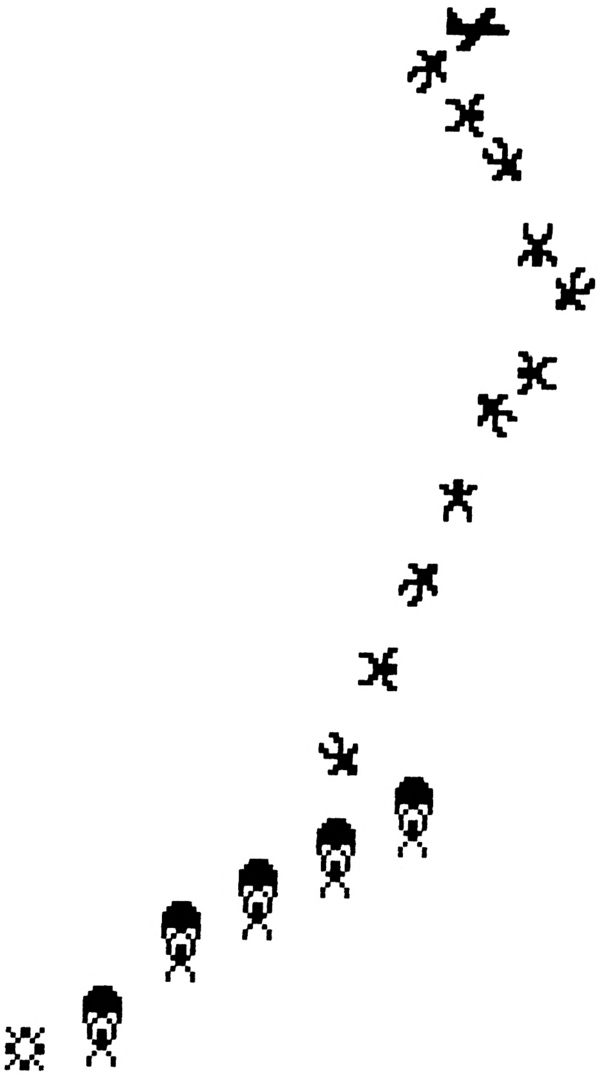


Fig 2.3 The free fall game

'ANOTHER GAME?' question subroutine 9500 will read key-presses until there aren't any left! Amstrad 664 users can use the CLEAR INPUT command to produce the same result.

The main program is composed of four WHILE loops. The outer loop, lines 40 to 290, simply keeps the game going as long as you answer 'yes' to the 'ANOTHER GAME?' question. The three inner loops correspond to the three stages of the game – lines 70 to 110 fly the plane before the down arrow key is pressed, lines

140 to 190 make the man fall until the COPY key is pressed and lines 220 to 270 make the man drift down under a parachute. The subroutines that are called before each of the WHILE loops initialise variables and generally get things ready for the next stage of the game. To make things easier, and to demonstrate the range of movement that can be created by making small changes, only one sprite is used. Its position, velocity and acceleration are given by X,Y,XV,YV,XA and YA respectively. Notice that it is only these quantities, and the shape of the sprite that are manipulated to produce the moving plane, the falling man and the parachuting man.

```
10   REM free fall game
20   MODE 1
30   GOSUB 2000
40   WHILE A$="Y"
50    GOSUB 1000
60    GOSUB 2500
70    WHILE INKEY(2)=-1
80     GOSUB 3000
90     GOSUB 5000
100    GOSUB 9000
110   WEND
120   GOSUB 2700
130   GOSUB 9500
140   WHILE Y<20 AND INKEY(9)=-1
150    T%=T%+1
160    GOSUB 9000
170    GOSUB 3000
180    GOSUB 4000
190   WEND
200   GOSUB 2900
210   GOSUB 9500
220   WHILE Y<=23
230    GOSUB 6000
240    GOSUB 3000
250    GOSUB 7000
260    GOSUB 9000
270   WEND
280   GOSUB 8000
290  WEND
999  END

1000 REM set up
1010 INK 0,0
1020 INK 1,26
1030 PEN 0
1040 PAPER 1
```

```
1050 CLS
1060 XL=10+RND*10
1080 LOCATE XL,25
1090 PRINT CHR$(238)
1999 RETURN

2000 REM init
2010 SYMBOL 240,&18,&DB,&7E,&18,&3C,&66,&42,&42
2020 SYMBOL 241,&33,&1F,&E,&7E,&CB,&89,&18,&30
2030 SYMBOL 242,&3,&E6,&34,&1F,&1F,&34,&E6,&3
2040 SYMBOL 243,&30,&18,&89,&CB,&7E,&E,&1F,&33
2050 SYMBOL 244,&42,&42,&66,&3C,&18,&7E,&DB,&99
2060 SYMBOL 245,&C,&18,&91,&D3,&7E,&70,&F8,&CC
2070 SYMBOL 246,&C0,&67,&2C,&F8,&F8,&2C,&67,&C0
2080 SYMBOL 247,&CC,&F8,&70,&7E,&D3,&90,&18,&C
2100 DIM S$(8)
2110 FOR I%=1 TO 8
2120  S$(I%)=CHR$(239+I%)
2130 NEXT I%
2140 SYMBOL 248,&3C,&7E,&FF,&FF,&FF,&99,&A5,&42
2150 SYMBOL 249,&5A,&5A,&3C,&18,&24,&42,&42,&0
2160 SYMBOL 250,&80,&C1,&E3,&FF,&FF,&E,&1C,&38
2170 SYMBOL 251,&C0,&C0,&80,&F0,&F8,&0,&0,&0
2180 P$=CHR$(250)+CHR$(251)
2190 A$="Y"
2499 RETURN

2500 REM plane init
2510 X=1
2520 Y=1
2530 XV=1
2540 YV=0
2550 XA=0
2560 YA=0
2570 XP=39
2580 YP=1
2599 RETURN

2700 REM drop init
2710 XV=0
2720 YV=0.5
2730 XA=RND*0.2-0.1
2740 YA=0.1
2799 RETURN

2900 REM para init
2910 YV=0.15
2920 XA=XA*3
2999 RETURN
```

```
3000 REM update
3010 XP=X:YP=Y
3020 X=X+XV
3030 Y=Y+YV
3040 IF X>38 THEN X=1
3050 IF XP>38 THEN XP=1
3060 IF X<1 THEN X=38
3070 IF XP<1 THEN XP=38
3080 XV=XV+XA
3090 YV=YV+YA
3999 RETURN

4000 REM reprint
4010 R=T% MOD 8+1
4020 CALL &BD19
4030 LOCATE XP,YP:PRINT "  ";
4040 LOCATE X,Y:PRINT S$(R);
4999 RETURN

5000 REM print plane
5010 LOCATE XP,YP
5020 PRINT "  ";
5030 LOCATE X,Y
5040 PRINT P$
5999 RETURN

6000 REM inkey
6010 IF INKEY(8)=0 THEN XV=-ABS(XV)
6020 IF INKEY(1)=0 THEN XV=ABS(XV)
6999 RETURN

7000 REM print para
7010 LOCATE XP,YP
7020 PRINT " ";
7030 LOCATE XP,YP+1
7040 PRINT " ";
7050 LOCATE X,Y
7060 PRINT CHR$(248);
7070 LOCATE X,Y+1
7080 PRINT CHR$(249);
7999 RETURN

8000 REM end game
8010 GOSUB 9500
8020 CLS
8030 IF ABS(X-XL)>3 THEN GOTO 8500
8040 PRINT "WELL DONE"
8050 IF ABS(X-XL)>1 THEN GOTO 8600
8060 PRINT "BRILLIANT, IN FACT"
8070 GOTO 8600
```

```
8500 PRINT "STAY ON THE GROUND"
8510 PRINT "IT'S SAFER"
8600 INPUT "ANOTHER GAME, Y/N";A$
8610 A$=UPPER$(A$)
8999 RETURN

9000 REM pause
9010 FOR W%=1 TO 250
9020 NEXT W%
9099 RETURN

9500 REM clear inkey
9510 WHILE INKEY$<>""
9520 WEND
9530 RETURN
```

To make the game more exciting and to check that you understand all of the ramifications of the sprite method you might like to add an erratic wind that blows the man off target all the time he is parachuting. There is also scope for the addition of some good sound effects for the aeroplane, missing the target and for the wind.

Conclusion

The sprite is the fundamental idea that lies behind all animation. However, as will be described in the next chapter, it is necessary to use all sorts of 'tricks' to create effective animation. This should not be taken to mean that the sprite is in fact useless. A program constructed using sprite ideas and a few special methods is still easier to write and understand than one that uses nothing but special methods – animation without sprites is like trying to do arithmetic without numbers!

Advanced animation 3

If computers were infinitely fast there would never be any need to consider any other form of animation than sprites. As it is, computers are rather slower than we would like and there is no choice but to examine some of the tricks that can be used to speed up the process of animation. Before going on to look at these tricks, it is worth looking at some of the more advanced features of the sprite idea.

High resolution sprites – TAG

All of the sprite examples in the previous chapter have used character graphics to make sure that everything moved reasonably fast. Using character graphics the smallest movement that a sprite can make is a single character location and while this results in fast movement it isn't very smooth. There is no reason however why a sprite cannot be animated using the high resolution co-ordinate system to record its position. As already described in Chapter One, following the TAG command, characters are printed with their top left hand corner at the current location of the graphics cursor. This allows the creation of sprites that move by a single graphics increment, and hence smooth, if not fast, motion. If you would like to see high resolution sprites in action then change the first version of the bouncing balls program given in Chapter Two as follows

```
1050 X(I%)-INT(RND*631)
1060 Y(I%)-INT(RND*191)+8

1090 XV(I%)-SGN(RND-0.5)*2
1100 YV(I%)=SGN(RND-0.5)*4

2080 IF X(I%)<=0 THEN X(I%)=0:XA(I%)=-2*XV(I%)
2090 IF X(I%)>=623 THEN X(I%)=623:XA(I%)=-2*XV(I%)
2100 IF Y(I%)<=15 THEN Y(I%)=15:YA(I%)=-2*YV(I%)
2110 IF Y(I%)>=399 THEN Y(I%)=399:YA(I%)=-2*YV(I%)

3010 MOVE PX(I%),PY(I%)
3020 TAG:PRINT SPC(1);:TAGOFF
3030 MOVE X(I%),Y(I%)
3040 TAG:PRINT S$(I%);:TAGOFF
```

The only really difficult part of the change is working out what co-ordinate values correspond to a sprite touching one of the boundaries. For example you might think that the position X,0 corresponds to a sprite on the lower edge of the screen but because this is the location of the top left hand corner of the character it actually corresponds to a position almost on the other side of the boundary! The y co-ordinate that corresponds to printing a character so that it is just completely on the screen is of course 15, the character height in terms of the high resolution co-ordinates. Using similar arguments, you should be able to work out the values used in line 2080 to 2110 to detect a collision. The only other point worth noticing is the way TAG and TAGOFF are used around the PRINT statements in lines 3020 and 3040 to alter the way text is printed to the screen. You could set TAG at the start of a program and leave it set all the way through but it is better to set and reset it as and when you need it. The reason for this is that it is better to have the machine in its usual state of separate text and graphics cursors while a program is running so that messages can be printed and control codes used.

This is all there is to high resolution animation, but there is the added bonus of being able to use the graphics ink modes – XOR, AND and OR. How these might be used is discussed as part of the following sections.

Multi-coloured sprites

For most applications the standard single colour sprite is quite sufficient, but occasionally the need to have more than a single colour does arise. As a character definition only involves a foreground and a background colour you might think that the requirement of more than one foreground colour rules out the use of characters to define a sprite shape. In fact it is very easy to combine characters to give any number of foreground colours and a single background colour. The key is the use of the transparent paper mode to overlay a range of foreground colours. For example if you want to print a white face with green eyes and a pink mouth you have to define three characters, a face, the eyes and the mouth, and then print them in the correct order after setting transparent paper mode and remembering to change the foreground colours. That is

```
10  MODE 0
20  SYMBOL 240,&00,&00,&66,&00,&00,&00,&00,&00
30  SYMBOL 241,&00,&00,&00,&00,&42,&3C,&00,&00
40  INK 1,26
50  INK 2,18
60  INK 3,16
70  PRINT CHR$(22);CHR$(1);
80  LOCATE 10,10
90  PEN 1
100 PRINT CHR$(224);
110 LOCATE 10,10
120 PEN 2
130 PRINT CHR$(240);
140 LOCATE 10,10
150 PEN 3
160 PRINT CHR$(241);
170 END
```

If you have an Amstrad 664 then you can set transparent paper without recourse to control codes by using an extra parameter in the PEN statement. (PEN L,1 sets transparent paper mode and PEN L,0 sets normal mode both using logical colour L.) However there are advantages in using control codes because they can be incorporated into a string and allow the multi-coloured sprite to be printed using a single PRINT statement. For example, if you make the following changes to the previous program

```
70  S$=CHR$(22)+CHR$(1)
        +CHR$(15)+CHR$(1)+CHR$(224)
        +CHR$(15)+CHR$(2)+CHR$(8)+CHR$(240)
        +CHR$(15)+CHR$(3)+CHR$(8)+CHR$(241)
        +CHR$(22)+CHR$(0)
80  LOCATE 10,10
90  PRINT S$
100 END
```

then the three-coloured face will be printed in a single operation by line 90. The first two control codes in the string set transparent paper, CHR$(15) sets the pen colour and CHR$(8) is a backspace command that makes sure that all three characters are printed in the same place. The final two control codes set the paper mode back to normal.

The same technique of using more than one character can be applied to high resolution sprites, that is following TAG, but you cannot use the control codes. Instead you have to set the graphics ink mode to OR and print each character at the same position by using the MOVE command.

Backgrounds

So far, the least interesting routine in the sprite animation loop has been the 'blank sprites' routine. All this does is to print or plot a uniform block of background dots to remove a sprite from the screen. However, this only works if the background against which the sprite are moving is itself a uniform colour! In most cases sprites do not move across a uniform background. For example, in a space game the sprites would move against a background of stars or even planets. If you simply blank sprites by printing blanks then it is obvious that the background will slowly be eroded as the sprites move around the screen. Commercial video games machines often get round this difficulty by using printed transparent overlays that are stuck to the front of the screen. However, this is not a method that is suitable for home computer use.

You might think that the simplest solution to the problem is somehow to save the area of the screen that the sprite is about to be printed at and then the next time though the animation loop the blanking subroutine could restore it. This is a possibility, but it requires two special machine code subroutines, one to 'get' an area of the screen and store it and another to 'put' the data back to the screen. In text mode there is a command only available on the 664 that will read the character already on the screen at a given location. The function COPYCHR$ returns a string containing the character at the position of the text cursor – if the character isn't recognised a null string is returned. This function can obviously be used to discover what is on the screen before a sprite is printed so that it can be used to blank the sprite out when it moves on. However the COPYCHR$ function isn't fast and, even though it is possible to add it to the 464's commands, a better method is to keep a screen copy in an array. For example, if you are using a mode 1 screen all you have to do is dimension an integer array DIM S%(40,25) and each time you PRINT a character to the screen at X,Y store its ASCII code in S%(X,Y) – that is for each LOCATE X,Y:PRINT CHR$(A); also do a S%(X,Y)=A. If these simple rules are followed, the array is a true copy of what is on the screen and can be examined at any time to find out what character a sprite is about to obliterate.

These methods do work but it doesn't solve the additional problem of what colour or pattern the sprite's own background dots should be. As already mentioned a sprite's shape is determined as a pattern of both foreground and background dots in a square array of dots. If the sprite's dots simply replace the dots on the screen then the patterned background will be destroyed even in places where it should show through. The solution is to store only the sprite's foreground dots in the screen area leaving the existing background dots as they are. This corresponds to using the transparent paper mode as described in the last section. In general, preserving backgrounds during sprite printing and blanking is not easy.

Blanking methods
Following the discussion of the problem of preserving backgrounds as sprites move across them it is worth looking at other methods of blanking. The most direct method of removing a character from the screen is to print a blank or space with the paper colour set correctly. However you will find that this method will fail if you have the transparent paper mode set and in this case what you should do is to print a solid block CHR$(143) using the correct pen colour.

In high resolution animation there is another way to blank a character using the XOR ink mode. If you print or draw anything on the screen while the XOR ink mode is set, then drawing it a second time erases it from the screen. For example try

```
10 MODE 1
20 PRINT CHR$(23);CHR$(1);
30 MOVE 100,100
40 TAG:PRINT "A";:TAGOFF
50 MOVE 100,100
60 TAG:PRINT "A";:TAGOFF
```

Line 20 sets the XOR mode, line 40 prints the letter 'A' which appears normally and then line 60 prints it in the same place a second time which causes it to disappear. Thus, using XOR ink mode, a character can be used to blank itself out with the advantage that the screen is returned to its original state. This method of blanking does preserve the background that the sprite moves over but it has the disadvantage that the sprite might not appear in the colour that you intended. For example if you print a sprite using logical colour 3 onto a background that has a logical colour of 2 using the XOR mode then the sprite will appear in logical colour 1 (3 XOR 2=1). This colour change is sometimes a considerable disadvantage of the method but for some applications it can actually help to make the sprite stand out against any background.

Finally it is worth mentioning the idea of the 'self blanking' sprite. If a sprite is going to move in only one or two known directions then it can be made to automatically blank out its old image by surrounding it with blanks. For example if a man-shape is only going to move across the screen from left to right one character location at a time it can be made self blanking by including a single space character to its left

```
10 MODE 1
20 S$=" "+CHR$(249)
30 FOR X=1 TO 39
40  LOCATE X,10
50  PRINT S$;
60 NEXT X
```

If you would like to see another more complicated example of a self blanking sprite then examine the example of velocity control of a bat in the previous chapter.

Synchronisation

A TV picture is built up as a number of lines, each scanned in turn starting from the top of the screen. If you change the character printed at a particular location then what you see depends on exactly when the change occurs. For example if the TV line scan has just displayed the first four rows of pixels of the old character before you print the new character then you will momentarily see half of the old character and half of the new character – producing a sort of flash. As all sprite movement is produced by repeated printing of characters on the screen this flashing will occur very frequently as the sprite moves across the screen.

The correct solution to this problem is not to change the creen while the TV screen is being scanned but to wait until the scanning spot is making its way back to the top of the screen ready for the next scan – this is called the 'vertical retrace period'. If you only print characters during the vertical retrace period there will be no danger of fractional characters being displayed and hence no flashing. There is a machine code routine at location

&BD19 that will wait until the start of the next vertical retrace period and the Amstrad 664 also has an additional command, FRAME, that does the same thing. If you are writing in machine code then you should always try to arrange to call &BD19 before starting to print anything to the screen and this should ensure flash free animation. However in BASIC the time it takes to execute a single command can be longer than the entire vertical retrace period and here the use of the CALL &BD19 or FRAME commands is less obvious. For example, you might be surprised by the results of the following program

```
10  MODE 1
20  X=20
30  Y=1
40  LOCATE X,Y
50  CALL &HBD19
60  PRINT " ";
70  IF INKEY(0)=0 THEN Y=Y-1
80  IF INKEY(2)=0 THEN Y=Y+1
90  LOCATE X,Y
100 CALL &HBD19
110 PRINT CHR$(143):
120 GOTO 40
```

This program is a simple animation loop that allows you to move a solid block up and down the screen. The only new feature is the use of CALL &BD19 (Amstrad 664 users can substitute a FRAME command) to synchronise the PRINT statements with the vertical retrace period. All that this should do is ensure that the animation is flicker-free but what actually happens is that for some vertical positions near the top of the screen the solid block changes its shape. The reason for this is simply that the PRINT statement takes longer to execute than the vertical retrace period. This means that the character is printed to the screen during the visible part of the scan. This, of course, can result in part of the old character being displayed along with part of the new character, but now rather than flashing randomly you get a stable image because the character is changed at the same point in the scan each time through the animation loop.

The moral of this story is that in BASIC the value of synchronisation is something that has not to be judged by trying it in each case.

Animating large objects

Animating large objects is not as common a requirement as you might imagine. This is fortunate, because in general it isn't easy!

The fact that sprites are small enough to make it possible tc 'quote' their dot patterns makes it easy to build a simple theor and method of using them. Large objects generally take too lon to draw to make it possible to use draw, blank and redrav animation loops in anything other than machine code. Even then it is often difficult to make things happen fast enough. The bes general method available is to make use of the observation that a a large object moves the dots that make it fall into three groups

1 those that remain unaltered
2 those that change to background dots, and
3 those that change to foreground dots

For most large objects, the set of dots that are unaltered at eac move accounts for most of the dots. This means that such a object can be animated by changing only the set of points tha should become background dots and the set of dots that shoul become foreground dots. For example, consider the problem o animating a large square block so that it moves horizontally acros the screen. As the block moves the trailing column of dots i changed to background dots and the leading column of dots i changed to foreground dots. The following program will animat quite a large block

```
10   MODE 2
20   X=0
30   GOSUB 1000
40   X=X+1
50   GOTO 30

1000 MOVE X,300
1010 DRAW X,200,0
1020 MOVE X+100,300
1030 DRAW X+100,200,1
1040 RETURN
```

Subroutine 1000 will draw a line in background dots betwee X,300 and X,200 and then a line in foreground dots betwee X+100,300 and X+100,200. At first all you will see is a movin line but as soon as the line of background dots reaches the firs line of foreground dots a moving square suddenly materialises!

This method of changing only the dots that need to be change sounds easy until you try it on a few apparently simple examples Finding out the sets of dots that change when a disk moves in horizontal line is bad enough but if it moves along a curved pat as in a simulation of a rising sun then things are really tricky!

Snake animation

There is one large object that can be easily animated using character graphics – the snake. If you look at a snake wiggling its way around the screen it looks as if all of the characters in its body move each time the snake moves. If this was the case animating a snake of any size would very quickly become a problem for assembly language. However, if all the characters that make up the snake are the same, only two characters – the head and the tail – actually need to move to give the impression that the whole snake is moving. The reason for this is not difficult to see. The head has to move because it is moving into a character location that was previously blank. On the other hand the second character in the snake has to move into the character location that the head occupied but if they are the same character there is actually no need to alter anything as long as we remember not to blank out the head at its old position. Applying this rule to each character down the snake it is obvious that nothing has to change until we reach the tail. In this case the argument about not having to move the tail into the position occupied by the last but one character holds, but in addition the old position of the tail has to be blanked out. What is surprising is that by printing only two characters, a head and a blank to erase the old tail position, an entire snake of any length will appear to move across the screen.

One common elaboration of this method is for the character that forms the head of the snake to be different from the rest of the body but even this amendment causes very little in the way of extra work. If the head is different from the rest of the body then its old position has to be changed to a character that forms the main body of the snake. So even a 'good looking' snake with a clear head needs only three characters to be printed to make it move, no matter how long it is.

Although the animation of a snake only requires the printing of a small number of characters each time through the animation loop there is still a problem in keeping track of the positions of all the characters in the snake. As moving the snake only involves the head and the tail you might be puzzled as to why you need to keep track of the positions of ALL of the characters in the snake. The reason for this is the need to know where the tail will be printed at each move. Each time the snake moves the tail moves to the position that was occupied by the last but one character and so it is clearly necessary always to know the position of the last but one character. But this argument can be repeated because each time the snake moves the second from last character becomes the last but one character in the snake! In other words, if all the co-ordinates of character I in the snake are stored in X(I) and Y(I) then at each move co-ordinates are up dated as follows

```
XT=X(1)
YT=Y(1)

FOR I=N TO 2 STEP -1
  X(I-1)=X(I)
  Y(I-1)=Y(I)
NEXT I
```

and

```
X(N)=XH
Y(N)=YH
```

where the snake is N characters long. The co-ordinates of the head are stored in X(N) and Y(N), XT and YT are the co-ordinates of the old position of the tail and XH and YH are the new co-ordinates of the head. If you examine this FOR loop you should be able to see that it moves all the co-ordinates down the array by one place – the co-ordinates of the first character in the snake become the co-ordinates of the second character and so on. After all the co-ordinates have been shifted, the animation of the snake is achieved simply by printing the head at its new position then changing the character at the head's old position to a snake 'body' character, and finally blanking out the old tail. The only trouble with the above method is that each time the snake moves the contents of the pair of arrays X(I) and Y(I) have to be shifted down. This is quite a lot of work for a BASIC program to do each time through the animation loop and, worse, the amount of work increases with the length of the snake. Using this method in BASIC a snake would move slowly and would grind (or slither?) to a virtual standstill as it grew in length. The solution to this difficulty is to be found in the use of an advanced 'data structure' known as a 'queue'. The idea is to avoid moving the data in the X and Y arrays by using a pair of pointers, one to the co-ordinates of the head and one to the co-ordinates of the tail. For example, if the co-ordinates of each character are once again stored in the arrays X and Y with Q being the index of the array elements that hold the co-ordinates of the head and Z being the index of the array elements that hold the co-ordinates of the tail, then the updating procedure becomes

```
Q=Q+1
X(Q)=XH
Y(Q)=YH

Z=Z+1
```

In other words, the new position of the head of the snake is stored one element further up the array than its old position. In this way, each time the head moves it leaves behind it a trail of the co-ordinates of its old positions. So, for example, if the head is currently at X(Q),Y(Q) then its previous position was X(Q−1), Y(Q−1), and before that it was at X(Q−2),Y(Q−2) and so on. This trail of co-ordinates can be used to make the tail follow the head around the screen simply by moving it from X(Z),Y(Z) to X(Z+1),Y(Z+1) each time through the animation loop. Technically the trail of co-ordinates and the variables Q and Z are referred to as a 'queue'. The term 'queue' seems appropriate if you think of the co-ordinates of the head as joining the end of a queue of co-ordinates and the co-ordinates of the new position of the tail forming the front of the queue. As the snake moves forward co-ordinates move down the queue to eventually become the current position of the tail as shown in this diagram

```
 ......|  |   |  |  |  |  |  |  |  ........
   old tail   ^              ^  unused
  positions   |              |  array elements
           current          current
           tail             head
           position         position
            X(Z),Y(Z)        X(Q),Z(Q)
```

This method of storing the co-ordinates of the head has one big problem – as the snake moves around the screen more and more array space is used up. However, at any one time only the elements between X(Z),Y(Z) and X(Q),Y(Q) are needed. The rest of the two arrays are either unused or hold old, now unwanted, tail positions. The solution is simply to make the array big enough to hold all of the co-ordinates of the longest snake that you are going to animate and, if either of the pointers Z or Q reaches the limit of the arrays, then reset them to 1. The best way to imagine the way that this works is to think of the arrays as being circular with their last element next to their first. In this sense the head and tail pointers Q and Z move round in a circle with all the co-ordinates of interest stored between them. With this small addition we now have all the ideas necessary to implement any number of games on animated snakes.

The game given in the listing below is perhaps the simplest of snake games but it still has the essential fascination involved in driving a snake around the screen. The game starts with a snake – actually a caterpillar in this case – surrounded by a mass of randomly placed flowers. The object of the game is to control the caterpillar (using the four cursor keys) so that it avoids all of the

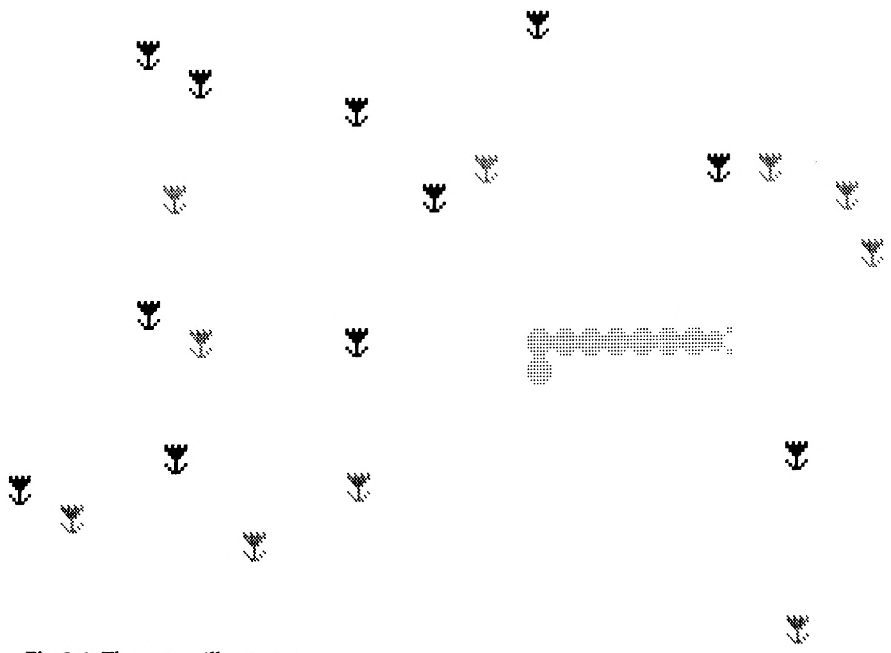


Fig 3.1 The caterpillar game

poisonous blue flowers and its own tail, but eats as many of the red flowers as possible. This starts off being quite easy but the catch is that the snake increases in length by one character for each red flower that it eats!

```
10    REM caterpillar
20    MODE 1
30    GOSUB 1000
35    GOSUB 1200
40    GOSUB 1500
90    WHILE LEFT$(A$,1)="Y"
100    GOSUB 1200
110    GOSUB 2000
120    WHILE FOOD%>0 AND DEAD%=0
130     GOSUB 3000
140     GOSUB 4000
150    WEND
180    GOSUB 9900
190   WEND
200   GOSUB 9600
210   GOSUB 9500
999   END

1000 REM init
1010 SYMBOL 240,&AA,&FE,&7C,&38,&10,&92,&54,&38
```

```
1020 SYMBOL 241,&3,&73,&FC,&F8,&F8,&FC,&73,&3
1030 SYMBOL 242,&C0,&CE,&3F,&1F,&1F,&3F,&CE,&C0
1040 SYMBOL 243,&C3,&C3,&24,&3C,&7E,&7E,&7E,&3C
1050 SYMBOL 244,&3C,&7E,&7E,&7E,&3C,&24,&C3,&C3
1060 SYMBOL 245,&3C,&7E,&FF,&FF,&FF,&FF,&7E,&3C
1070 DIM H$(4)
1080 FOR I%=1 TO 4
1090  H$(I%)=CHR$(240+I%)
1100 NEXT I%
1110 FOOD%=10+RND*5
1120 DEAD%=0
1130 DIM X%(30),Y%(30)
1140 A$="Y"
1150 MX%=30
1199 RETURN

1200 REM re_init
1210 Q%=5
1220 Z%=1
1230 FOR I%=1 TO 5
1240  X%(I%)=I%:Y%(I%)=10
1250 NEXT I%
1260 XH%=5:YH%=10
1270 A%=5:B%=10
1280 XV%=1:YV%=0
1290 H%=1
1490 RANDOMIZE TIME
1499 RETURN

1500 BORDER 26
1510 INK 0,26
1520 INK 1,3
1530 INK 2,11
1540 INK 3,2
1550 CLS
1999 RETURN

2000 REM plant flowers
2005 CLS
2010 PEN 1
2020 FOR I%=1 TO FOOD%
2030  GOSUB 2500
2040 NEXT I%
2080 PEN 2
2090 FOR I%=1 TO 5+RND*5
2100  GOSUB 2500
2110 NEXT I%
2150 PEN 3
2160 LOCATE X%(1),Y%(1)
2170 FOR I%=1 TO 4
2180  PRINT CHR$(245);
```

```
2190 NEXT I%
2200 PRINT CHR$(241);
2499 RETURN

2500 REM flower pos
2510 X=RND*39+1
2520 Y=RND*24+1
2530 U%=X:V%=Y
2540 GOSUB 9000
2550 IF CL%<>0 THEN GOTO 2510
2560 LOCATE X,Y
2570 PRINT CHR$(240);
2999 RETURN

3000 REM inkey
3010 IF INKEY(0)=0 THEN YV%=-1:XV%=0:H%=3
3020 IF INKEY(2)=0 THEN YV%=1:XV%=0:H%=4
3030 IF INKEY(8)=0 THEN XV%=-1:YV%=0:H%=2
3040 IF INKEY(1)=0 THEN XV%=1:YV%=0:H%=1
3050 GOSUB 9600
3999 RETURN

4000 REM print cat
4005 FOR W%-1 TO 100:NEXT W%
4010 A%-XH%:B%-YH%
4020 XH%=XH%+XV%:YH%=YH%+YV%
4030 IF XH%>40 THEN XH%=1
4040 IF XH%<1 THEN XH%=40
4050 IF YH%>25 THEN YH%=1
4060 IF YH%<1 THEN YH%=25
4070 U%=XH%:V%=YH%
4080 GOSUB 9000
4090 IF CL%<>0 THEN GOSUB 4500
4100 LOCATE A%,B%
4110 PRINT CHR$(245);
4120 LOCATE XH%,YH%
4130 PRINT H$(H%);
4140 Q%=Q%+1
4150 IF Q%>MX% THEN Q%=1
4160 X%(Q%)=XH%:Y%(Q%)=YH%
4170 LOCATE X%(Z%),Y%(Z%)
4180 PRINT " ";
4190 Z%=Z%+1
4200 IF Z%>MX% THEN Z%=1
4499 RETURN

4500 REM hit
4510 IF CL%<>1 THEN GOTO 4600
4520  PRINT CHR$(7)
4530  FOOD%=FOOD%-1
4540  Z%=Z%-1
```

```
4550  IF Z%<1 THEN Z%=MX%
4590 GOTO 4999
4600  SOUND 1,2000,50,15
4610  DEAD%=1
4999 RETURN

9000 REM point
9010 HU%=(U%-1)*16+8
9020 HV%=400-16*(V%-1)-8
9030 CL%=TEST(HU%,HV%)
9099 RETURN

9500 REM reset
9510 INK 0,0
9520 INK 1,26
9530 BORDER 0
9540 PEN 1
9550 PAPER 0
9560 CLS
9599 RETURN

9600 REM clear inkey
9610 WHILE INKEY$<>""
9620 WEND
9699 RETURN

9900 REM end game
9905 CLS
9910 IF DEAD%=1 THEN GOTO 9940
9920  PRINT "WELL DONE"
9930 GOTO 9950
9940  PRINT "BAD LUCK"
9950 INPUT "ANOTHER GAME Y/N";A$
9960 A$=UPPER$(A$)
9970 FOOD%=10+RND*5
9980 DEAD%=0
9999 RETURN
```

You should be able to recognise all of the elements of snake animation in the above program with the help of the following subroutine table

Subroutine	*action*
1000	initialise user-defined characters and dimension arrays
1200	initialise caterpillar variables
1500	set colours
2000	print flowers and first caterpillar

2500	print a single flower
3000	examine cursor keys
4000	update caterpillar's position
4500	caterpillar has hit something
9000	test screen colour
9500	reset paper and pen at end of game
9600	clear keyboard buffer
9900	end game

The only subroutine worth comment is 9000 which uses the TEST function and a conversion from text co-ordinates to high resolution co-ordinates to discover the colour of the middle of the character location that the caterpillar head is about to move into.

Colour mapping

The way that physical colours can be assigned to logical colours can be used to produce very fast and very smooth animation without recourse to the print/reprint cycle of blanking animation. If a number of areas are created using different logical colours and all but one of them is made invisible by assigning the background physical colour to them then, by changing which of the areas is assigned to the foreground physical colour, the illusion of movement can be created. This method of producing apparent movement is known as 'colour mapping'. As an example, the following program creates the appearance of two objects moving backwards and forwards across the screen

```
10   MODE 0
20   GOSUB 1000
30   GOSUB 2000
40   GOSUB 3000
50   GOTO 40

1000 FOR X=1 TO 15
1010  PEN X
1020  LOCATE X,16
1030  PRINT CHR$(143);
1040  LOCATE 16-X,20
1050  PRINT CHR$(143);
1060 NEXT X
1070 RETURN

2000 FOR L=1 TO 15
2010  INK L,0
```

```
2020 NEXT L
2030 RETURN

3000 FOR L=1 TO 15
3010  INK L-1,0
3020  INK L,26
3030  FOR J=1 TO 10:NEXT J
3040 NEXT L
3050 INK 15,0
3060 RETURN
```

Subroutine 1000 prints two rows of 'blocks' in such a way that each block in the row has a different colour. In fact, the logical colour increases as you move from left to right in the first row and decreases in the second row. Subroutine 2000 assigns all of the logical colours to physical colour zero thus hiding all of the blocks. Following this, subroutine 3000 repeatedly assigns each of the logical colours in turn to the physical colour white. In this way the block that is visible moves across the screen. As the assignment of physical to logical colours doesn't take very many memory accesses colour mapped animation can be very fast indeed.

As a second example the following program draws a sequence of circles in logical colours 1 to 15 and then animates them by assigning all but one to the background colour

```
10   MODE 0
20   A%=320
30   B%=199
40   C%=1
50   FOR R%=10 TO 200 STEP 20
60    C%=C%+1 MOD 16
70    GOSUB 1000
80   NEXT R%
90   GOSUB 2000
100  GOSUB 3000
110  GOTO 100

1000 INC=2/R%
1010 IC=COS(INC)
1020 IS=SIN(INC)
1030 X=R%:Y=0
1040 PLOT A%-X,B%+Y,C%
1050 PLOT A%+X,B%+Y
1060 FOR T=0 TO PI/2 STEP INC
1070  Z=X
1080  X=X*IC-Y*IS
1090  Y=Z*IS+Y*IC
```

```
1100  PLOT A%+X,B%+Y
1110  PLOT A%-X,B%+Y
1120  PLOT A%+X,B%-Y
1130  PLOT A%-X,B%-Y
1140 NEXT T
1150 RETURN

2000 FOR L%=1 TO 15
2010  INK L%,0
2020 NEXT L%
2030 RETURN

3000 FOR L%=1 TO 15
3010  INK L%-1,0
3020  INK L%,26
3030  FOR J=1 TO 50:NEXT J
3040 NEXT L%
3050 INK 15,0
3060 RETURN
```

Flash animation

Many programmers only think of using the flashing colours to draw attention to a particular area of the screen or for implementing an explosion, but they can be used as another method of animation based on colour – flash animation. The basic idea of using the flashing colours to produce the illusion of movement is to arrange the screen so that adjacent areas flash in the reverse order. That is, if a character location is flashing black then white, its next door neighbour would be flashing white and then black. In this way the flashing looks like something moving between the adjacent character locations. Perhaps the best way to understand this idea is by a simple example. The following program draws a series of 'blocks', each consisting of two character locations, to form a frame around the screen which is then animated using flashing colours. The blocks are written bright red and bright white respectively.

```
10   MODE 1
20   GOSUB 1000
30   GOSUB 2000
40   END

1000 REM PRINT FRAME
1010 FOR X=1 TO 40 STEP 4
```

```
1020   LOCATE X,1
1030   PEN 1
1040   PRINT STRING$(2,143);
1050   PEN 2
1060   PRINT STRING$(2,143);
1070   LOCATE X,24
1080   PEN 1
1090   PRINT STRING$(2,143);
1100   PEN 2
1110   PRINT STRING$(2,143);
1120  NEXT X
1130  FOR Y=2 TO 23
1140   PEN Y2 MOD 2 +1
1150   LOCATE 1,Y
1160   PRINT CHR$(143);
1170   LOCATE 40,Y
1180   PEN (Y2+1) MOD 2 +1
1190   PRINT CHR$(143);
1200  NEXT Y
1210  RETURN

2000  INK 1,6,26
2010  INK 2,26,6
2020  SPEED INK 8,8
2030  RETURN
```

Notice the use of SPEED INK to adjust the speed of flashing so that the illusion of motion is enhanced. Try different values in this statement, line 2020, to see the effects produced. Also notice that flashing animation has the strange property that it continues even when the program isn't running!

Scrolling animation
The final 'novel' method of animation is based upon the simple observation that the Amstrad, in common with many other computers, can move a great many pixels very quickly by means of a 'text scroll'. Obviously using simple scrolling you are limited to moving objects up the screen but by using text window feature it is possible to create a number of independent moving streams. For example the following program animates a central strip of the screen by first defining a narrow text window and then printing light and dark blocks on the bottom line, so forcing the window to scroll

```
10    MODE 1
20    GOSUB 1000
30    GOSUB 2000
40    GOTO 30
```

```
1000 FOR I=1 TO 10
1010  LOCATE INT(RND*39)+1,INT(RND*24)+1
1020  PRINT CHR$(249);
1030 NEXT I
1040 WINDOW #1,15,19,1,25
1050 C=0
1060 RETURN

2000 IF C=1 THEN C=0 ELSE C=1
2010 PEN#1,C
2020 LOCATE#1,1,25
2030 PRINT#1,STRING$(5,207)
2040 RETURN
```

Text window scrolling can be used to implement a wide range of games where something moves vertically past another sprite. It is possible to implement a horizontal scroll using one of the control registers in the Amstrad's graphics chip but a full discussion of this hardware is beyond the cope of this book. However the following program does illustrate the basic principles involved.

```
10   S%=0
20   D%=S% MOD 256
30   R%=13
40   GOSUB 1000
50   S%=S%+1
60   FOR J=1 TO 100:NEXT J
70   GOTO 20
100  GOTO 20

1000 OUT &BC00,R%
1010 OUT &BD00,D%
1020 RETURN
```

Subroutine 1000 sets register R% in the video controller to the value in D%. In this case S% is used to determine the horizontal offset of the screen and this produces the sideways scrolling effect. Try using this routine in a game to scroll a landcape background past a fixed object.

4 Charts and graphs – an introduction to co-ordinates

The idea of using a co-ordinate pair x,y to pick out a single pixel is so familiar that to most graphics programmers it is second nature. However picking out an image pixel by pixel would be a laborious task and much of computer graphics is concerned with finding ways of specifying whole collections of pixels without having explicitly to list the co-ordinates of each one. In this chapter we look at some of the ways that co-ordinates can be used to generate shapes, and programs to produce graphs and charts are used as examples. By way of a contrast in the next chapter we look at ways of making shapes without the direct use of co-ordinates.

Subroutines that draw lines

The most obvious way of writing programs that create graphics displays is to simply go ahead and write some BASIC that creates each element of the display. That is, if you are trying to draw a house then write some BASIC that draws the rectangle for the base of the house, then some BASIC that draws the roof, then the windows and so on until you have the desired display. If you do follow this simple method then you will find that your programs are very long, very repetitive and very difficult to modify. A much better way to program graphics is to write subroutines that generate particular shapes and then call these subroutines in the correct order. This approach is sometimes referred to as 'procedural graphics'. The advantage of doing things this way is that if you have written a subroutine that will draw a rectangle then it can be re-used each time you want a rectangle. Of course, to be really useful such a subroutine should allow you to specify where the rectangle should be drawn, its size and its orientation. As described in Chapter One the most important tool in creating such shape-drawing subroutines are the relative graphics commands.

Using the relative graphics commands you can design a shape without reference to where it is to be drawn. All you have to do is choose a single point in the shape and treat this as if it was located at 0,0. Then using DRAWR or PLOTR commands draw each line or point in the shape. If you then use a MOVE X,Y statement before all of the relative commands that produce the shape it will be drawn with the point that you treated as at 0,0 at X,Y. This recipe is in fact a lot easier to follow in an example. Suppose you want to draw a square with a side of 100 units. If you treat the lower left hand corner as 0,0 then the commands to draw the square are

```
1010 DRAWR 0,100
1020 DRAWR 100,0
1030 DRAWR 0,-100
1040 DRAWR -100,0
```

which can be thought of a drawing lines between the corners at 0,100 100,100 and 100,0. Now if you want to draw the square with its lower left hand corner at X,Y then add

```
1000 MOVE X,Y
```

It's as easy as that.

In other words, you should always design shapes with one corner located at the origin, using relative commands, and then move the shape to the position at which it is required using an absolute MOVE command.

All of this fuss about being able to move shapes to any position only begins to pay off if you also make use of subroutines along with location, size and orientation parameters. However in practice it turns out to be too difficult to write subroutines that can produce a given shape at any angle and usually we settle for varying the location and size of the shape. (For a partial solution to the rotation problem see 'chain coding' later in this chapter.) For example, to draw a rectangle you should write a subroutine something like

```
5000 REM rectangle H by W with
5010 REM bottom left corner at X,Y
5020 MOVE X,Y
5030 DRAWR 0,H
```

```
5040 DRAWR W,0
5050 DRAWR 0,-H
5060 DRAWR -W,0
5070 RETURN
```

then you never have to write lines of BASIC to draw a rectangle EVER again. In some senses a subroutine call like

```
X=10:Y=30:H=50:W=100:GOSUB 5000
```

which will draw a rectangle at 10,30 and 50 by 100, can be thought of almost as an addition to the BASIC graphics commands. By building up a collection of graphics subroutines of this sort you can eventually reduce the production of a new graphics program to nothing more than putting together the correct sequence of subroutine calls!

A bar chart

As an example of the use of procedural graphics, and as an introduction to the 'scaling' problem, consider the task of drawing a bar chart like the one shown in fig 4.1. The bar chart is obviously made up of two graphics elements – a pair of axes and a

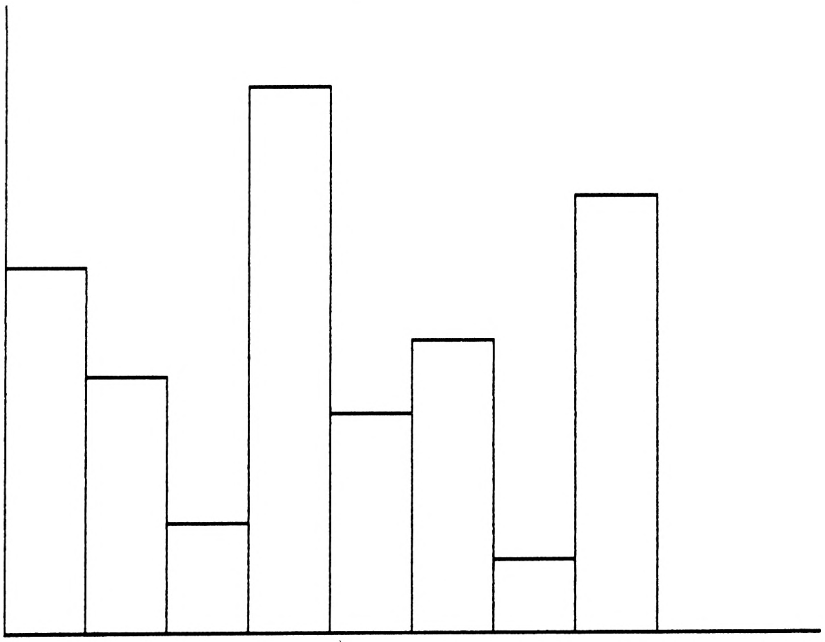


Fig 4.1 A bar chart

number of blocks of variable height. Writing subroutines for these is not difficult. What is difficult is making sure that the bar chart stays on the screen no matter what data has to be represented. If the largest data value is MX% and this has to be represented by a bar 300 units high to fill the screen then for each data unit the bar should be 300/MX% graphics units high. Simply multiplying each data value by 300/MX% gives the height of its corresponding bar and so 300/MX% is called a 'scaling factor'. The final program is

```
10    REM bar chart
20    MODE 2
30    GOSUB 1000
40    GOSUB 2000
50    GOSUB 3000
60    X=50:Y=50:GOSUB 4000
70    FOR I%=1 TO NUM%
80     X=50+(I%-1)*50
90     Y=50
100    H=P(I%)
110    W=50
120    GOSUB 5000
130   NEXT I%
140   WHILE INKEY$=""
150   WEND
199   END

1000 REM init
1010 NUM%=8
1020 MX%=0
1030 DIM D(NUM%),P(NUM%)
1999 RETURN

2000 REM data
2010 DATA 10,7,3,15,6,8,2,12
2020 FOR I%=1 TO NUM%
2030  READ D(I%)
2040  IF D(I%)>MX% THEN MX%=D(I%)
2050 NEXT I%
2999 RETURN

3000 REM stats calc
3010 SC=300/MX%
3020 FOR I%=1 TO NUM%
3030  P(I%)=D(I%)*SC
3040 NEXT I%
3999 RETURN

4000 REM axes
4010 CLG
4020 MOVE X,Y
```

```
4030 DRAWR 500,0
4040 MOVE X,Y
4050 DRAWR 0,500
4060 RETURN

5000 REM bar
5010 MOVE X,Y
5020 DRAWR 0,H
5030 DRAWR W,0
5040 DRAWR 0,-H
5999 RETURN
```

Subroutine 1000 initialises variables, NUM% is the number of data values. Subroutine 2000 reads the data values from a DATA statement into the array D and also finds the maximum value and stores it in MX%. Subroutine 3000 calculates the scaling factor SC% and works out the height of each bar in the array P%. Subroutine 4000 plots a set of axes and subroutine 5000 draws a single bar of width W and height H with its bottom left hand corner at X,Y. The main program simply uses subroutine 5000 to draw each bar in turn. You can replace subroutine 2000 by one to read in data of your own.

Sets of pixels – functional representation

The use of subroutines to generate simple shapes works well but so far we can only use it to generate shapes composed of straight lines. There is clearly a need for some way of creating curves. As already mentioned the fundamental graphics object is the pixel and a single pixel can be specified by quoting its co-ordinates, but quoting the co-ordinates of single pixels is not a practical method of specifying the curves that we need. Somehow we have to find a way of automatically generating the co-ordinates of pixels that form the standard curves that we use. To take the most obvious example, it is clearly unreasonable to store the co-ordinates of each point on a straight line. What we need is an algorithm that will generate the co-ordinates of each point on the line as needed. Of course there is a well known algorithm for this in the form of the 'equation of a line'

$$y=mx+c$$

where m and c are constants that govern the angle of the line and its position on the screen respectively. This equation effectively connects the values of the x and y co-ordinates for points that make up the line. For example, if m=2 and c=3 then the equation is

y=2x+3

and if x is 1 then y=2*1+3 or 5 and the point 1,5 is part of the line. Notice that this only works because for every value of x there is a value of y that gives a point that is part of the line. As x is varied between 0 and XMAX a set of points is generated (not all of which are guaranteed to lie on the screen) that form a straight line.

This idea of using an equation to generate sets of points that make up a curve is easy to generalise. In essence all you need is an equation that connects x and y. This is usually written as

y=f(x)

which means that for a given value of x the function 'f' can be used to calculate a value of y that gives a point x,y, that is part of the curve. As another example

y=SQR(r^2−x^2)

will, if SQR generates the positive square root, pick out all the points that form the upper half of a circle centered on the origin, and with radius r. In this case the equation only makes sense for values of x that lie between −r and +r. Outside this range the SQR function is trying to take the square root of a negative quantity and will generate an error.

This way of specifying sets of pixels is known as 'functional representation'. Although functional representation looks as if it might be the key to all graphics problems it is in fact remarkably difficult to find functions which generate any particular curve. For example, you would search a long time before you found the equation that generated the shape of a motor car! In practice there are only three really useful curves that are specified by simple functions: the straight line, the circle and the ellipse. These curves can be used in combination to build up larger and more complicated shapes. In fact, most computer graphics involves nothing but straight lines to construct outline shapes is described but before moving on it is worth examining functional representation and the problem of drawing curves in more detail.

A graph plotter

As equations of the form

y=f(x)

are so important it is worth developing a program that will draw the graph of any such equation. The basic idea behind such a program is simply to generate a range of values of x, calculate y= f(x) for each one and then plot the points x,y. The only problem is once again scaling – that is making sure that the graph stays on the screen. As in the case of the bar chart it is important to make sure that the point corresponding to the largest value of y is on the screen. In addition we also have to take care that the smallest value of y (y can now be negative) and the full range of x can be displayed on the screen. If the largest and smallest values of Y are YMAX and YMIN then

$$\frac{(Y-YMIN)}{(YMAX-YMIN)}$$

is a quantity that varies between 0 and 1 (try Y=YMIN and Y= MAX to see that this is true). Therefore

$$\frac{(Y-YMIN)*400}{(YMAX-YMIN)}$$

varies between 0 and 400 and so corresponds to the full range of vertical screen co-ordinates. By the same sort of argument

$$\frac{(X-XMIN)*640}{(XMAX-XMIN)}$$

varies between 0 and 640 and so corresponds to the full range of horizontal screen co-ordinates. These two functions are called 'scaling' functions and can be used to make any range of values correspond to either the x or y co-ordinate range. Using these functions the program is fairly easy to write, the values of XMIN and XMAX can be obtained from the user and the values of YMIN and YMAX can be obtained by working out the function for all the values of X to be plotted and finding the smallest and largest values of Y. The only question is how many values of X between XMIN and XMAX should be plotted? The answer is that each value of X should differ by an amount that moves the plotted point on by one screen pixel. So the X step size should be

$$S=\frac{(XMAX-XMIN)*XINC}{640}$$

where XINC is the graphic increment (see Chapter One) for the mode in use. If mode two is used then XINC is 1 and the equation is simply

$$S=\frac{(XMAX-XMIN)}{640}$$

After all this discussion the program should now be easy to understand

```
1     DEF FNE(X)=SIN(X)+SIN(2*X)

10    REM graph plot
20    MODE 2
30    GOSUB 1000
40    GOSUB 2000
50    CLG
60    GOSUB 3000
70    WHILE INKEY$=""
80    WEND
99    END

1000 REM init
1010 INPUT "Draw graph starting at X=";XMIN
1020 INPUT "Last X value to be graphed=";XMAX
1999 RETURN

2000 REM scale
2010 X=XMIN
2020 YMIN=FNE(X)
2030 YMAX=YMIN
2040 S=(XMAX-XMIN)/640
2050 FOR X=XMIN TO XMAX STEP S
2060  Y=FNE(X)
2070  IF Y>YMAX THEN YMAX=Y
2080  IF Y<YMIN THEN YMIN=Y
2090 NEXT X
2999 RETURN

3000 REM graph
3010 FOR X=XMIN TO XMAX STEP S
3020  Y=FNE(X)
3030  PLOT (X-XMIN)/(XMAX-XMIN)*640,
     (Y-YMIN)/(YMAX-YMIN)*400
3040 NEXT X
3999 RETURN
```

Subroutine 1000 asks the user to supply the values of XMAX and XMIN, subroutine 2000 finds YMAX and YMIN and finally subroutine 3000 plots the graph using the scaling equations given above. The equation that is plotted is specified as the user-defined function FNE an example of which is given as line 1 above. To generate other curves, change line 1 to specify other equations. The following are a few of the infinite possibilities:

x*x
x*x*x
SIN(x)

A sample of the output can be seen in fig 4.2.

Parametric curves
The idea of using a function to generate a straight line or curve is easy to understand and use, but it has its limitations. For example you cannot generate closed curves using functional representation without using a number of different functions, each generating a part of the curve. The reason for this is simple to see once you realise that for a closed curve there is more than one value of y for each value of x (see fig 4.3) and a single function can only return one value of y for each value of x. A more powerful method of representing curves is to use a function for each

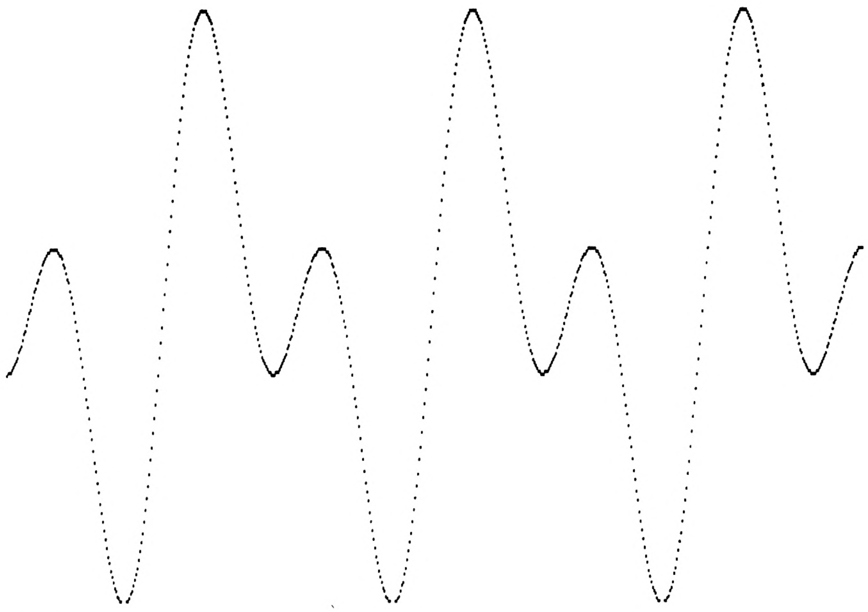


Fig 4.2 Graph plot

co-ordinate. That is, instead of using a function that generates a value of y given a value of x, i.e. y=f(x), use two functions

x=f1(t)
y=f2(t)

that generate a value of x and y given a value t. Using a function to generate each co-ordinate of a curve is known as 'parametric representation'. The variable 't' is referred to as the 'parameter' and all of the points that lie on the curve are generated as t varies between an upper and a lower limit. Many interesting and complicated curves are surprisingly simple to generate using a parametric representation. For example, the functions

```
X=A*(3*COS(T)+COS(3*T))
Y=A*(3*SIN(T)-SIN(3*T)))
```

generate a curve called an 'astroid' as T varies between −PI and +PI. The variable A is simply a constant that controls the size of the astroid. The following program will draw a sequence of astroids using the above functions

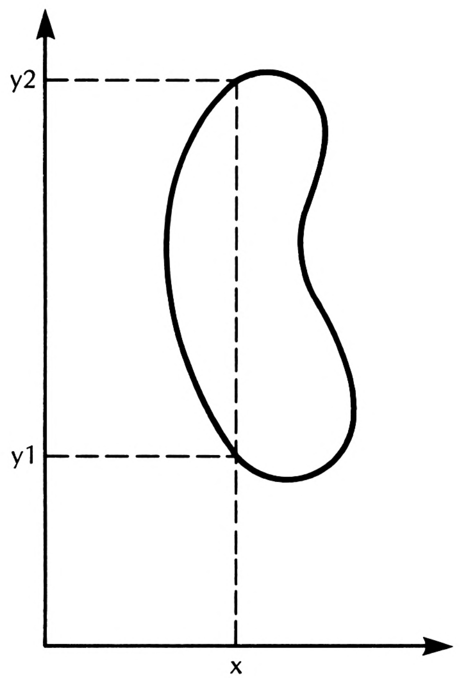


Fig 4.3 A closed curve has more than one value of y for each value of x

```
10  MODE 2
20  FOR A=1 TO 4
30   FOR T=-PI TO PI STEP .1
40    X=A*(3*COS(T)+COS(3*T))
50    Y=A*(3*SIN(T)-SIN(3*T))
60    X=X*10+320
70    Y=Y*10+200
80    IF T=-PI THEN MOVE X,Y ELSE DRAW X,Y
90   NEXT T
100 NEXT A
120 END
```

Lines 60 and 70 simply scale the co-ordinates so that the astroids fit on the screen. A sample of the output of this program can be seen in fig 4.4. It is worth pointing out that there is a functional representation of the astroid but it is very complex.

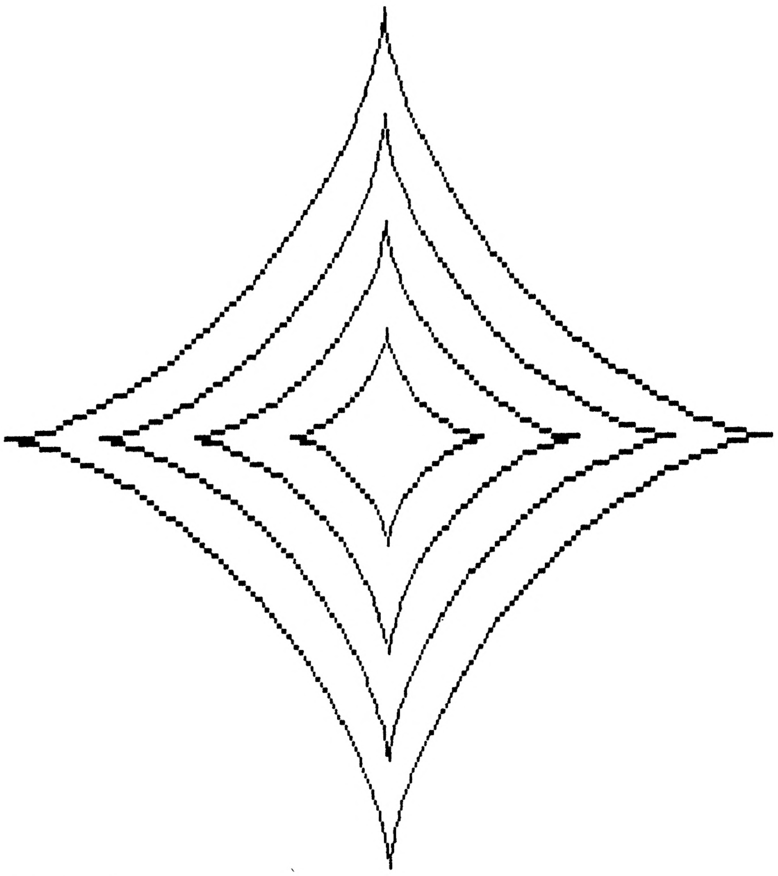


Fig 4.4 Astroid

There are a great many interesting curves that can easily be generated using parametric representation but very few of them are useful for anything other than forming abstract patterns. Even though parametric representation is not the whole solution to the problem of drawing curves it does lead to an efficient method of drawing two of the most important closed curves, the circle and the ellipse, and it is fundamental to a method of drawing arbitrary curves, 'interpolation'.

Practical circles

Although it isn't worth going into great detail about how to draw a straight line from first principles – the DRAW command makes this an academic exercise – it is worth developing subroutines to draw circles and ellipses. Many versions of BASIC come equipped with circle and ellipse drawing commands but these have been omitted from Amstrad BASIC. Rather than just quoting a pair of finished subroutines to make up for this shortcoming the details of how a pair of parametric equations are turned into efficient subroutines is described as a short graphics study.

The parametric equations for the circle are

X=R*COS(T)+A
Y=R*SIN(T)+B

where R is the radius, the point at A,B is the centre and T varies from 0 to 2*PI. The point X,Y moves completely around the circle as T goes from 0 to 2*PI and this is the origin of the 'radian' measure of angle. 2*PI is a full circle or 360 degrees, PI is half a circle or 180 degrees and so on. These equations are very easy to turn into a procedure that will draw a circle

```
10   MODE 2
20   FOR R%=10 TO 200 STEP 10
30    A%=320:B%=199:C%=1:GOSUB 1000
40   NEXT R%
50   END

1000 INC=.1
1010 FOR T=0 TO 2*PI  STEP INC
1020  X%=R%*COS(T)
1030  Y%=R%*SIN(T)
1040  PLOT X%+A%,Y%+B%,C%
1050 NEXT T
1060 RETURN
```

If you run this program you will discover that while it does indeed plot the outline of a circle on the screen there are gaps in the circumference, and the size of the gaps increases with the size of the circle. The reason for this behaviour is not difficult to work out. The number of points plotted per circle is fixed by the value of INC but it is obvious that the circumference of a large circle needs more points than the circumference of a small circle. In other words, INC should get smaller as the radius of the circle increases. After a little algebra, or by trial and error, it is not difficult to discover that line 1000 should be changed to

```
1000 INC=1.0/R%
```

Following this change all of the circles drawn by the subroutine have continuous circumferences and so are satisfactory. However the number of SIN and COS evaluations needed for each circle, especially the large ones, makes the subroutine very slow so the next thing to look at is efficiency.

Whenever you need to calculate SIN and COS at a number of evenly spaced values there are ways of reducing the amount of work involved. The key equations are the 'sum' formulae for SIN and COS

COS(A+B)=COS(A)*COS(B)−SIN(A)*SIN(B)
SIN(A+B)=SIN(A)*COS(B)+COS(A)*SIN(B)

where A and B are angles. If you look back at the FOR loop in subroutine 1000 you will see that it is calculating SIN and COS of an angle that increases by INC each time through the loop. So if COS(T) and SIN(T) are the values obtained the last time through the FOR loop the new values are

COS(T+INC)=COS(T)*COS(INC)−SIN(T)*SIN(INC)
SIN(T+INC)=SIN(T)*COS(INC)+COS(T)*SIN(INC)

by use of the 'sum' formulae given earlier. If you examine this equation you will notice that COS(INC) and SIN(INC) are constant and so they can be worked out just once before the loop starts and then used each time through to update the old values of SIN and COS. Starting from the initial values of COS(0) and SIN(0), which equal 1 and 0 respectively, the FOR loop can update both COS and SIN each time through the loop using nothing but simple arithmetic. The 'sum' formulae can be used to reduce the amount of work whenever a series of SIN or COS values is required. This is a great saving in the amount of calculation required to draw a circle but in this case there is still one more simplification. On the Ith time through the FOR loop the X and Y values calculated are

Xi=R*COS(T)
Yi=R*SIN(T)

the next time through the loop, that is the i+1th time, the X and Y values calculated are

Xi+1=R*COS(T+INC)
Yi+1=R*SIN(T+INC)

Using the 'sum' formulae to expand the COS and SIN functions gives

Xi+1=R*COS(T)*COS(INC)−R*SIN(T)*SIN(INC))
Yi+1=R*SIN(T)*COS(INC)+R*COS(T)*SIN(INC))

If you compare these expressions with those for Xi and Yi you will see that

Xi+1=Xi*COS(INC)−Yi*SIN(INC)
Yi+1=Yi*COS(INC)+Xi*SIN(INC)

In other words, each time through the FOR loop the new co-ordinates can be obtained from the old co-ordinates by multiplication and addition involving a pair of constants COS(INC) and SIN(INC). We can also take advantage of the symmetry of a circle to reduce the work involved in drawing it by three quarters. If A+X,B+Y is a point on the circle then so are A−X,B+Y A+X,B−Y and A−X,B−Y and this can be used to plot the full circle from points calculated from the first quarter, (see fig 4.5). Putting all of these ideas into practice give the following replacement for subroutine 1000 given earlier

```
1000 INC=1/R%
1010 IC=COS(INC)
1020 IS=SIN(INC)
1030 X=R%:Y=0
1040 PLOT A%-X,B%+Y
1050 PLOT A%+X,B%+Y
1060 FOR T=0 TO PI/2 STEP INC
1070  Z=X
1080  X=X*IC-Y*IS
```

```
1090  Y=Z*IS+Y*IC
1100  PLOT A%+X,B%+Y
1110  PLOT A%-X,B%+Y
1120  PLOT A%+X,B%-Y
1130  PLOT A%-X,B%-Y
1140 NEXT T
1150 RETURN
```

Notice the use of the variable Z in line 1070 to store the value of X for use in calculating the update to the Y co-ordinate in line 1030. If you run this subroutine along with the main program given earlier you will see the same set of circles – but much more quickly!

A pie chart

As an example of the use of the circle drawing program, and of the knowledge of how a circle is drawn, consider the problem of writing a program to draw a pie chart like that shown in fig 4.6. The problem of drawing the circle is now easily solved using the subroutine given in the previous section. The problem of dividing it up into segments that have areas in the same ratio as the data values is a little more tricky. If there are three data values, say 50, 30 and 20, then the pie chart should have three segments with areas equal to 50%, 30% and 20% of the entire circle. Working out

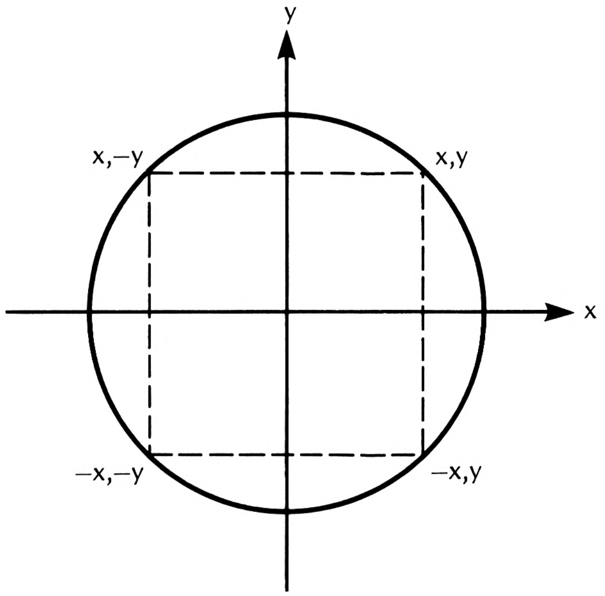


Fig 4.5 Circle showing the use of symmetry

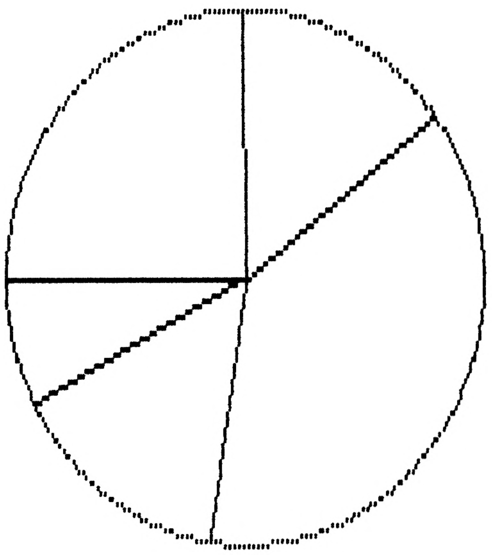


Fig 4.6 Pie chart

segment areas is a little difficult however and it is easier to use the fact that the areas of the segments are in the same ratios as the segment angles. So, in the previous case, if the segment angles are 180, 108 and 72 degrees then the segments will have the correct areas. In other words if a data value is given by D, and the total of all the data values is TT, then the angle of the segment that represents it is given by

$$T=360*\frac{D}{TT} \quad \text{degrees}$$

or

$$T=2*PI*\frac{D}{TT} \quad \text{radians}$$

You can think of 2*PI/TT as a sort of scaling factor that converts the data values into segment angles between 0 and 2*PI. The reason that the segment angles are easier to use than areas is simply that if you want to draw a radius at angle T from the start of the circle all you have to do is use the pair of commands

```
MOVE A%,B%
DRAW A%+R%*COS(T),B%+R%*SIN(T)
```

The final program is

```
10   REM pie chart
20   MODE 2
30   GOSUB 1000
40   GOSUB 3000
50   GOSUB 4000
60   GOSUB 2000
70   GOSUB 5000
199  END

1000 REM init
1010 R%=100
1020 A%=320
1030 B%=199
1040 C%=1
1050 MX%=5
1060 DIM D(MX%),P(MX%)
1070 T=0
1080 TT=0
1999 RETURN
2000 REM circle
2010 INC=2/R%
2020 IC=COS(INC)
2030 IS=SIN(INC)
2040 X=R%:Y=0
2050 PLOT A%-X,B%+Y
2060 PLOT A%+X,B%+Y
2070 FOR T=0 TO PI/2 STEP INC
2080  Z=X
2090  X=X*IC-Y*IS
2100  Y=Z*IS+Y*IC
2110  PLOT A%+X,B%+Y
2120  PLOT A%-X,B%+Y
2130  PLOT A%+X,B%-Y
2140  PLOT A%-X,B%-Y
2150 NEXT T
2999 RETURN

3000 REM data
3010 DATA 10,3,6,15,6
3020 FOR I%=1 TO MX%
3030  READ D(I%)
3040  TT=TT+D(I%)
3050 NEXT I%
3999 RETURN

4000 REM stats calc
4010 N=2*PI/TT
4020 FOR I%=1 TO MX%
4030  P(I%)=D(I%)*N
```

```
4040 NEXT I%
4999 RETURN

5000 REM draw segs
5030 FOR I%=1 TO MX%
5040  T=T+P(I%)
5050  MOVE A%,B%
5060  DRAW A%+R%*COS(T),B%+R%*SIN(T)
5065  FOR J=1 TO 250:NEXT J
5070 NEXT I%
5999 RETURN
```

Subroutine 1000 initialises all of the variables and sets up the necessary arrays; subroutine 2000 is the circle drawing routine given at the end of the previous section; subroutine 3000 reads the data in and finds its total in TT; subroutine 4000 works out the scaling factors and segment angles and finally subroutine 5000 draws the radius lines in at the correct angles.

Ellipses

Now that we have a subroutine that draws circles the next step is to extend it. Perhaps the most useful shape after a circle is an ellipse and this can be drawn using similar techniques. The parametric equations for an ellipse (with its major or minor axis horizontal) are simply

X=R1*COS(T)+A
Y=R2*SIN(T)+B

where the ellipse is centered on A,B, and R1 and R2 are the horizontal and vertical 'radii' of the ellipse. Using the same methods as for the circle it is not difficult to work out that

Xi+1=Xi*COS(INC)−Yi*SIN(INC)*R1/R2
Yi+1=Xi*SIN(INC)*R2/R1−Yi*COS(INC)

Using these equations to update the co-ordinates each time through the FOR loop gives the following program

```
10    MODE 2
20    FOR E-1 TO 2 STEP .2
30     A%-320:B%-199:C%-1:R1%-100:R2%-E*R1%:
       GOSUB 1500
40     A%=320:B%=199:C%=1:R2%=100:R1%=E*R2%:
       GOSUB 1500
50    NEXT E
60    END
```

```
1500 REM ellipse
1510 IF R1%>R2% THEN INC=1.0/R1% ELSE INC=1.0/R2%
1520 IC=COS(INC):IS=SIN(INC)
1530 IS12=IS*R1%/R2%:IS21=IS*R2%/R1%
1540 X=R1%:Y=0
1550 PLOT A%+X,B%+Y
1560 FOR T=0 TO 2*PI STEP INC
1570  Z=X
1580  X=X*IC-Y*IS12
1590  Y=Z*IS21+Y*IC
1600  PLOT A%+X,B%+Y
1610 NEXT T
1620 RETURN
```

the output of which can be seen in fig 4.7. Once again a speed improvement is possible if the symmetry of the ellipse is used but this is left as an exercise for the reader.

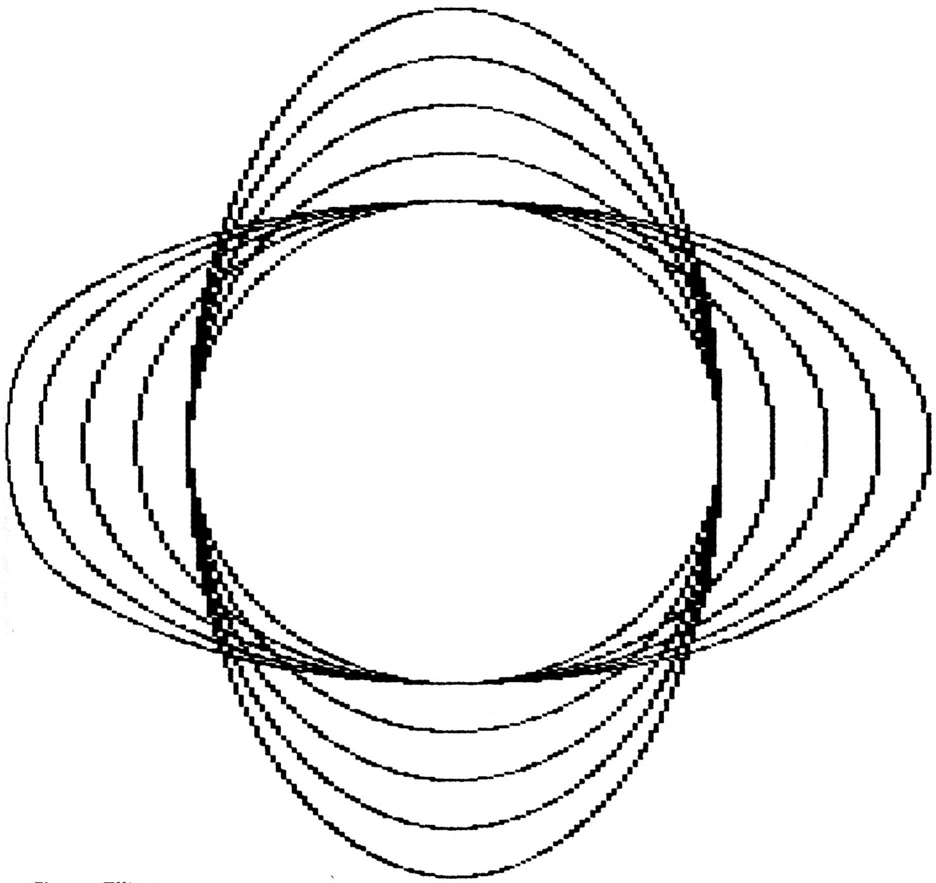


Fig 4.7 Ellipse

The final subroutines given for drawing circles and ellipses are still capable of improvement – for example you could change the circle program so that it will draw an arc between two giver points – but they are good enough to be used if an assembly language subroutine is not available. The idea of relating the new co-ordinates to the previous co-ordinates is a very general one ir graphics and you should always look out for such possibilities.

Chain code – a way of rotating shapes

There is an alternative way of describing the shape of an outline in terms of straight line segments that has advantages when it comes to scaling and rotation – chain coding. The fundamental idea of chain coding is to specify a starting position, using the usual X,Y co-ordinate system, and then describe the shape ir terms of movements from the current position. For example, a square could be described as

start at X,Y
then move 10 units up
then, from your new position, move 10 units to the left
then 10 units down and 10 units right

You can work out a chain code for any shape without too much trouble and once you have done it, the description itself forms a program that produces the shape! In addition all this is possible without reference to a co-ordinate system, apart from fixing the starting position of the outline. You should be able to see that, ir this form, chain coding is very similar to the way that the relative graphics commands were used at the start of this chapter. However if the chain code is represented in a symbolic form it car be easily converted to a recipe for drawing a shape at different angle.

The best way to illustrate the ideas behind chain coding is to give a program that will interpret a simple chain code. The following program recognises four movement commands U, D, L and R, each of which can be followed by a number. The meaning of the commands is obvious: U means up, D means down, L means left and R means right. You can type collections of commands into the program and each will be obeyed in turn. For example, U10R10D10L10 will draw a small square. As well as the four movement commands the program will also obey a rotation command of the form 'An' where A stands for Angle and n is 1, 2, 3 or 4. Following An all shapes specified by a chain code will be drawn rotated by an angle of n*90 degrees. This sounds like something that is difficult to implement until you realise that a rotation of 90 degrees can be produced by simply changing U to R, R to D, D to L and L to U in any chain code. In the same way a rotation of any multiple of 90 degrees can be produced by a permutation of the letters standing for the movement commands. If commands that produce lines at 45 degrees are added, then rotations of any multiple of 45 degrees can be produced using the same method. For example if you type in the following program and enter the chain code U100R50D50L50R50D50 then you will see the shape of the letter A. If you enter the same chain code a second time but preceded by A1 then you will see a letter A on its side and so on.

```
10   MODE 2
20   GOSUB 1000
30   GOSUB 2000
40   GOSUB 3000
50   GOTO 30

1000 C$="URDL"
1010 MOVE 320,199
1020 R=0
1030 RETURN

2000 LOCATE 1,25
2010 PRINT SPC(79);
2020 LOCATE 1,25
2030 PRINT "CHAIN CODE ";
2040 INPUT S$
2050 RETURN

3000 I=1
3010 WHILE I<=LEN(S$)
3020  GOSUB 9000
3030  D$=MID$(S$,I,1)
3040  L=VAL(MID$(S$,I+1,LEN(S$)))
3050  A=INSTR(1,C$,D$)
3060  IF A<>0 THEN A=A+R
3070  IF A>4 THEN A=A-4
3080  GOSUB 4000
3090  IF D$="A" THEN GOSUB 7000
3100  I=I+1
3110 WEND
3120 RETURN

4000 IF A=1 THEN GOSUB 5000
4010 IF A=2 THEN GOSUB 5500
4020 IF A=3 THEN GOSUB 6000
```

```
4030 IF A=4 THEN GOSUB 6500
4040 RETURN

5000 DRAWR 0,L
5010 RETURN

5500 DRAWR L,0
5510 RETURN

6000 DRAWR 0,-L
6010 RETURN

6500 DRAWR -L,0
6510 RETURN

7000 R=L
7010 RETURN

9000 WHILE ASC(MID$(S$,I,1))<57 AND I<LEN(S$)
9010  I=I+1
9020 WEND
9030 RETURN
```

The program works by scanning the chain code stored in the string S$ and finding each single letter command in turn. The current position in the string is stored in I and subroutine 9000 will add one to I until it is 'pointing' at a letter rather than a number. (That is, subroutine 9000 is a 'find the next letter in the string' procedure.) This letter is then stored in D$ and the VAL function is used to convert the digits following it into a numeric value stored in L. The command letter in D$ is then searched for in C$, which is a sort of 'command table'. If the letter is found its position in C$ is used to decide which of the four movement subroutines should be called. However if the letter is an 'A' then subroutine 7000 is called, which simply sets the variable R to the value stored in L. The reason for doing this is that R is added to the position returned by the INSTR function and so 'shifts' the apparent positions of the commands in the command table C$ as required to implement a rotation. For example if R is 0 and the command is 'U10' then the letter stored in D$ is 'U', its position in C$ is 1 and so subroutine 5000 is called. However if R has been set to 1 by a 'A1' command then, even though 'U' is still the first letter in C$, subroutine 5500 is called as if the command had been 'R10'. If you study the program you should be able to add commands for movement in 45 degrees and 45 degree rotations. Other features that you might like to add include an absolute move command to position the graphic cursor at X,Y, a scaling command that alters the size of a shape by simply setting a multiplier for L and a colour setting command.

Chain codes are a useful way of dealing with some graphics problems that involve shapes that have to be printed anywhere, at any size and with a rotation that is a multiple of 90 or 45 degrees. However they still don't solve the problem of drawing something rotated through any given angle – a solution to the more general problem is given in Chapter Six.

5 Freestyle graphics – painting

Co-ordinates are such a standard way of doing things that it can be difficult to imagine that there is another approach to graphics. In Chapter Four the basic methods of using co-ordinates to draw regular shapes were described and these are very successful if the object being drawn is either regular or can be accurately measured. So, if you are interested in the exact representation of a building, a floor plan or an electronic circuit diagram then co-ordinate based methods are the best. However, if you are interested in creating original artwork then methods based on the direct manipulation of the screen display, so called 'co-ordinate free' methods, are far more suitable.

This idea of working directly with the patterns that the pixels form is a difficult idea, at first, because we are so used to programming in terms of co-ordinates. The best way of seeing the essence of the approach is to think about what happens when you draw with a more traditional technology – pencil and paper. Unless you are using a drawing board for technical drawing, most pencil sketches are not carried out in terms of co-ordinates. You draw rough lines and shade in areas without measuring their exact position. If something doesn't look right then you erase it and modify it. In this sense you are 'moulding' the patterns of dark and light directly. If you change the technology used with this approach to light pen and screen then the same direct interaction with the pixels that form the image is possible – however, surely the computer can offer something more than just the pencil and paper operations of marking and erasing? The answer is most certainly 'yes' but you might be surprised at the amount of computer power necessary to achieve it.

In practice, using entirely 'co-ordinate free' methods turns out to be an unnecessary burden. The best compromise is often to use co-ordinates as a way of implementing an idea that is essentially co-ordinate free. However sometimes it has to be admitted that some things are best done using a co-ordinate approach. For example it is difficult to think of a better way of drawing a straight line than using a functional equation to join two end points. The co-ordinate free approach is often more a matter of spirit rather than technique – it could eqully well be called the 'artistic' approach to computer graphics.

Pointing at points

The fundamental data unit underlying all graphics is the point or pixel. Obviously, if we are going to draw on the screen then there must be some way of entering information about pixels. The only truly co-ordinate free method of doing this is the light pen, or similar device, that allows you physically to point at the pixel or pixels that you want to change. Unfortunately, light pens are not popular or common input devices and so a co-ordinate based method has to be used. The most obvious method of selecting a point on the screen is to use a visible marker for the current position of the graphics cursor. How this visible graphics cursor is moved around the screen determines how acceptable this pixel selection method is. For example, the worst possible way of controlling the graphics cursor is to use the four cursor keys to update its position. This produces a very difficult-to-use system that tends to stifle creativity. A much better method is to use a joystick to supply x,y co-ordinates directly. The trouble with this method is that the Amstrad's joysticks are not true proportional joysticks, they are simply extensions of the keyboard. Thus using the Amstrad joysticks for co-ordinate free graphics has no real advantage over using the keyboard. For this reason the programs given in the rest of this chapter use the cursor keys to control position. This is not ideal, and if you do have either a true proportional joystick or a mouse then it is worth changing the programs to use it. A mouse is a small box with a pair of wheels or a ball bearing that allows it to be pushed around on the table top next to the computer. As the mouse is moved it sends signals to the computer to tell it how far in the horizontal and vertical direction it has moved. In this sense using a mouse is very similar to using the cursor keys to update the position of the graphics cursor but in practice it is very much easier to use.

Depositing pixels – the brush

Programming a graphics cursor that moves around the screen according to which cursor key is pressed is relatively easy. The only complication is that, to avoid the cursor modifying any image already on the screen, it has to be drawn using the XOR graphics ink mode. When the XOR graphics ink mode is selected, printing or drawing something once makes it appear on the screen, and printing or drawing it a second time at exactly the same position removes it and restores the screen to its original condition. Once you have a graphics cursor under your control you can use it to specify any pixel on the screen – but what next! You could set things up so that pressing a key on the keyboard would change the pixel that the graphics cursor was currently over to a specified colour. Using this idea you could change one pixel at a time – clearly drawing lines and shapes is going to take a long time! Looking back at more traditional drawing methods suggests that an approach that might be worth trying is to copy the action of a paint brush or a pencil. As a paint brush is drawn across the paper or canvas it leaves a trail or paint behind it. This is easy to implement. When a key, COPY say, is pressed any pixel that the graphics cursor passes over will be changed to the selected colour.

If you try this out what you will find that it is very difficult to produce anything but 'spidery' sketches. The trouble is that it is like trying to draw with an extremely fine pencil or brush. The idea is good but we haven't implemented the 'brush' idea with sufficient flexibility. A brush should clearly affect more than one pixel at a time – but how many? The answer is that we need a range of 'brushes' varying in thickness and perhaps even in shape. The characteristics of a brush are defined by the shape that it produces when it isn't moved around the screen. For example, in fig 5.1 you can see a circular brush and a rectangular brush. If

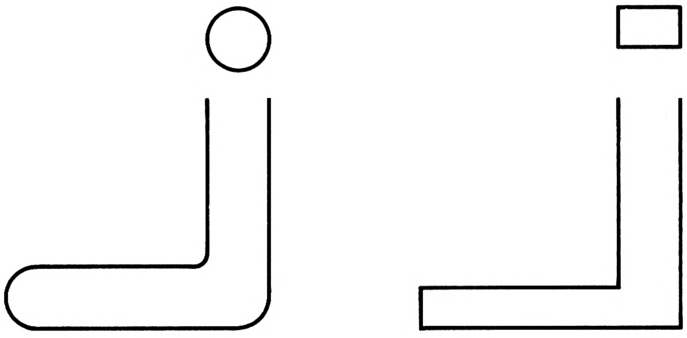


Fig 5.1 The results of using a round and a square brush

you were to move the graphics cursor to a position and select a one of the three brushes then it would produce the shape shown in fig 5.1. Moving the graphics cursor with the brush still selected would leave a trail of brush patterns behind it. The circular brush will allow you to draw lines of equal thickness in all directions. The rectangular brush will produce lines of different thickness depending on which way it is moving. Obviously each brush has to be supplied in a range of sizes to enable the thickness of the line to be controlled. Notice also that erasing can be done by painting with a brush set to the local background colour.

Simple brushes can easily be implemented as user-defined characters on the high resolution screen, that is printed following a TAG command. For example, if you want a round brush then all you have to do is to define a character in the shape of a disk. This character can then be used as a marker for the current position of the graphics cursor. If the character is printed using the XOR graphics ink mode it will appear to move around the screen in response to the cursor keys without affecting anything. However if the graphics ink mode is changed to OR, in response to pressing the COPY key say, then printing the character becomes a permanent change to the screen and moving it leaves a trail behind it exactly as required for a brush.

Given the basic concept of the software generated brush then all sorts of ideas spring to mind for more sophisticated brushes, for example, texture brushes. Instead of changing every point within the brush's area it could affect a pattern of points. A random pattern would give a stippling brush and wavy lines would make drawing a seascape really easy! The idea of a brush is clearly worth examining in more detail.

Functional brushes

All of the brushes described in the last section have one thing in common – they all deposit 'paint' in a way that depends only on their shape and where they are on the screen. A functional brush is once again an obvious extension of the brush idea but it includes the extra facility of looking at what pixels are already on the screen before depositing any 'paint'. For example, suppose that by accident you had 'splashed' some foreground pixels into a region of the screen. You could remove them by taking a very fine brush and touching them out. However, you could use a brush that examines each pixel within its area and changes it to the colour of the majority of its neighbours. By passing such a 'clean up' brush across the splashed area a few times it should be possible to get rid of any mess without damaging what was already present before the splash. Another example of a functional brush is one that will 'sharpen corners'. It can be difficult to draw accurately at the best of times and many corners in an image will be ragged, as shown in fig 5.2. To sharpen these corners all that is needed is a brush that will introduce the missing pixel where a row and a column of pixels 'cross', as in fig 5.2. Passing this brush over the image will only change the points in question. One of the great advantages of a functional brush is that, using it, you don't run the risk of accidentally damaging any part of the image that you have already completed.

Pattern brushes

It is easy to apply areas of even colour using a traditional or a computer paint brush, but filling an area with a regular pattern is difficult. For example, suppose that you had just drawn the outline of a house and wanted to draw a brick pattern on the walls. The only way that this could be achieved using a simple brush is to paint in each brick in turn! Clearly one of the ways that computer art could be made easier than traditional art is by the provision of a brush that will deposit paint in a regular pattern, that is a 'pattern brush'.

A pattern brush is clearly a special case of a functional brush but it is very special in that the colour of every pixel on the screen is determined by its position on the screen. You might think that all you have to do to implement a pattern brush is to define a character which produces the pattern when printed and then use it as a simple brush as described erlier. If you try this you will discover that all you get is a solid trail left behind the brush as it moves. The reason for this is that the foreground points in the brush set each screen pixel they pass over irrespective of the pattern that you are trying to produce.

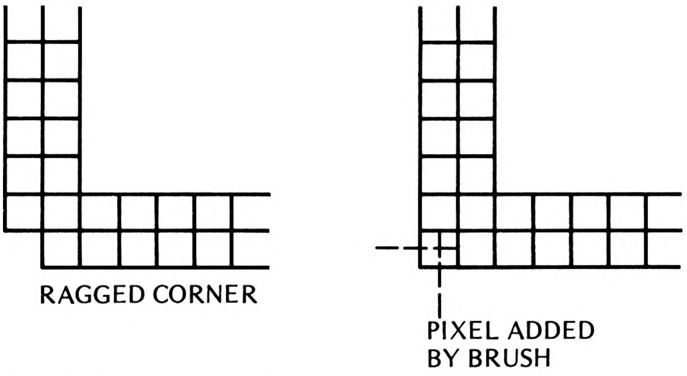


Fig 5.2 The use of a brush to 'touch up' a corner

The correct way to produce a pattern brush using characters involves defining a character that produces that appropriate pattern at each location on the screen. To understand this idea it is helpful to imagine a pattern, diagonal lines say, printed across the entire screen. As a brush is moved across this screen the portion of the pattern that passes under the brush is exactly what the brush has to produce if it is to paint the pattern on a blank screen. If you think about it for a moment you should be able to see that this implies that the character printed at each location has to change according to the location. This sounds like far too much work for a program, especially a BASIC program to do in a reasonable amount of time but if the pattern repeats itself over a small area of the screen the number of different characters that you need is reduced to manageable proportions. One method of producing pattern brushes is decribed in detail later as part of AMART, an interactive painting program.

Scan filling

The problem of filling an outline with a given pattern has now been solved as long as the artist using the pattern brush is skillful enough to keep within the outline! Filling areas with even tones or patterns is time-consuming and prone to mistakes. This makes it an ideal candidate for improvement as a computer implementation. A 'scan fill' brush automatically produces a given pattern within the confines of an outline. All that the user has to do is to move a marker to a position within the outline and then press a key, COPY say, and sit back and watch the outline fill with the pattern.

There are two distinct methods of filling an outline – 'scan fill' and 'flood fill'. The difference is that only the flood fill method guarantees to fill an outline without leaving any unfilled areas. As you might expect scan fill is much easier to program and the areas that it leaves are easy to deal with by subsequent scan fill operations.

As an example of a scan fill, consider the problem of filling a circle with a colour to produce a disk. Starting from a given point x,y the procedure is to fill the circle one horizontal line at a time. Starting from x,y the pixels to the right of this point are examined in turn until the first non-background pixel is found. (Recall that the function TEST(x,y) can be used to discover the logical colour of the pixel at x,y). Then using this pixel as a starting point a similar scan to the left is performed, once again looking for the first non-background pixel. The only difference is that during this left scan each pixel that is tested is subsequently set to the foreground colour – thus drawing a line from the right hand edge of the circle to the left hand edge. Once this line is complete the entire line scan is repeated one line down, that is using x,y−2 as the starting point and so on down the screen. This downward scan is stopped when the starting point for the line scan is found to be not a background pixel. When the downward scan is complete an upward scan is performed, so filling the top half of the circle. If you are doubtful about the details of this process then the best way to clarify what happens is to watch the circle being filled by the following program

```
10    MODE 2
20    R%=100:A%=320:B%=200:GOSUB 1000
30    A%=320:B%=200:GOSUB 2000
40    END

1000 INC=1/R%
1010 IC=COS(INC)
1020 IS=SIN(INC)
1030 X=R%:Y=0
1040 PLOT A%-X,B%+Y
1050 PLOT A%+X,B%+Y
1060 FOR T=0 TO PI/2 STEP INC
1070  Z=X
1080  X=X*IC-Y*IS
1090  Y=Z*IS+Y*IC
1100  PLOT A%+X,B%+Y
1110  PLOT A%-X,B%+Y
1120  PLOT A%+X,B%-Y
1130  PLOT A%-X,B%-Y
1140 NEXT T
1150 RETURN

2000 XT%=A%:YT%=B%
2010 WHILE TEST(XT%,YT%)=0
2030  GOSUB 3000
2040  XT%=XT%-1
2050  GOSUB 4000
2060  YT%=YT%-2:XT%=A%
2070 WEND
2080 XT%=A%:YT%=B%+2
2090 WHILE TEST(XT%,YT%)=0
2100  GOSUB 3000
2110  XT%=XT%-1
2120  GOSUB 4000
2130  YT%=YT%+2:XT%=A%
2140 WEND
2150 RETURN
```

```
3000 WHILE TEST(XT%,YT%)=0
3010  XT%=XT%+1
3020 WEND
3030 RETURN

4000 WHILE TEST(XT%,YT%)=0
4010  PLOT XT%,YT%,1
4020  XT%=XT%-1
4030 WEND
4040 RETURN
```

Subroutine 1000 is the circle drawing program given in the previous chapter. Subroutine 2000 performs the scan fill, starting from the point A%,B%. It calls subroutine 3000 to perform the left scan and subroutine 4000 to perform the right scan.

After seeing the results of the scan fill program given above you might find it difficult to imagine a case where an area is left unfilled. As an example of this add the following line to the previous program

```
25 R%=40:A%=370:B%=200:GOSUB 1000
```

As you can see in fig 5.3 the new circle blocks the line scan to the right and this results in a sort of shadow effect. If this scan fill routine was part of an interactive painting program then all the artist would have to do is to perform a second scan fill with the starting point somewhere within the unfilled area.

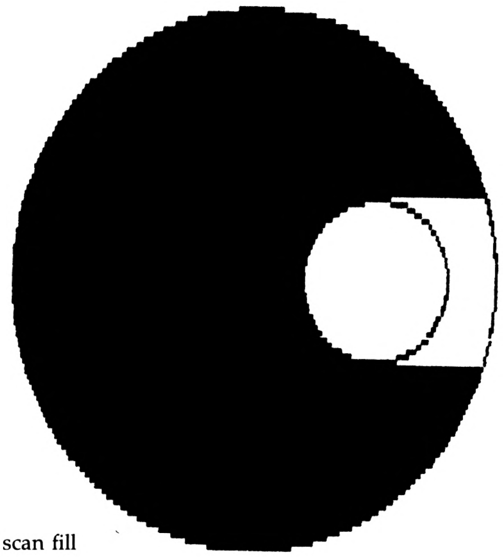


Fig 5.3 The effect of a scan fill

With a simple modification, the scan fill program can be made to fill outlines with a specific pattern. All that is needed is a test that determines the colour of a point that is about to be plotted as part of the left line scan. The details of how this is done are given as part of the painting program described below.

Users of the Amstrad 664 have access to a special flood fill command which will do a better job than the scan fill subroutine given above. The command

FILL L

will perform a flood fill starting from the current position of the graphics cursor. If you would like to see how this differs from the scan fill, change line 30 in the previous demonstration program to read

```
30 A%=370:B%=200:MOVE A%,B%:FILL 1
```

In both cases you will find that FILL is much quicker than subroutine 2000 and in the second case it also succeeds in filling the entire circle. However, the FILL command does have the disadvantage that it cannot be used to fill an area with a regular pattern and so it will not be used as part of the interactive painting program given later in this chapter.

Electric pencils

Even though brushes in one form or another serve for most of the tasks in computer assisted art, there is still the occasional need to draw a straight line. In this case, what we are interested in is a computer equivalent of the ruler and pencil. While it would be possible to create something that was the software equivalent of a ruler and pencil a better method of drawing straight lines interactively is based on the 'rubber band' principle. Rubber banding is a general technique for drawing lines and shapes that will be described in more detail in the next chapter. However drawing a line using rubber banding is quite simple. One end of the line is considered fixed at a specified position and the other end is drawn to the current position of the graphics cursor. If the line is drawn using the XOR graphics ink mode and blanked out just before each move of the graphics cursor, the effect produced mimics a rubber band stretched between a fixed point and the moving graphics cursor. All that is needed to turn this into a good way of drawing lines is some way of specifying the position of the fixed point and some way of making the line permanent once it has just been stretched into its desired position. To enable a series of joined up lines to be drawn the best way of moving the fixed point is to have its position transferred to the current position of the graphics cursor when a specific key, F7 (keypad 7) say, is pressed. To make the line permanent it simply has to be drawn with the OR graphics ink mode set, once again in response to a specific key press, the COPY key say. Using this method of control, a line could be drawn between any two points by first moving the graphics cursor to the first point, pressing F7 to move the fixed point to the position of the graphics cursor and then moving the graphics cursor to the second point on the line. All the time that the graphics cursor is being moved the line is being re-drawn between the fixed point and the position of the graphics cursor so giving the user feedback on how the line looks. After making this line permanent by pressing the COPY key a second line can be drawn connected to the first by pressing F7 and then moving the graphics cursor. It is amazing how easy this method of drawing lines is. If you have any difficulty in understanding the explanation then simply try the pencil option in the interactive painting program given below.

As well as drawing lines by rubber banding it is possible to draw all manner of shapes. For example, to draw a rectangle using rubber banding all you have to do is use the same idea of a fixed point and a moving point but in this case they determine opposite corners of the rectangle rather than the two ends of a line. By continuously re-drawing the rectangle as the graphics cursor is moved, the user can see it change and interactively adjust its size and position. A rubber band rectangle drawing option is also included in the interactive painting program given below.

An interactive painting program – AMART

Most of the ideas introduced in this chapter have been incorporated into a single large program – AMART. Although it is written entirely in BASIC it includes a surprising number of features. It includes a pattern brush in a range of sizes and shapes, an eraser, a scan fill brush, a line drawing pencil and a rectangle drawing option. It also includes the facility to save and load pictures using disk or tape and printer dump option. No attempt has been made to use colour. The program paints in black on a white background, the emphasis being on drawing in high resolution. If you want to introduce colour then you will have to change the program to work in MODE 1 and add coloured patterns. This is not at all difficult in theory but you might find that the low resolution and small screen size results in an unsatisfactory program.

Rather than use commands typed in from the keyboard to control the program, an 'icon' based menu (one that uses symbols to represent choices) has been used as an example of how graphics can be used as to create a user-friendly environment. Indeed it is probably easier to learn to use the program by using it than by reading this description! When the program is run the screen is cleared and a number of menu boxes are printed down the right hand side, see fig 5.4. To select one of the options all the user has to do is move the graphics cursor over the menu box that contains either the relevant symbol (icon) or command word and then press the COPY key. The menu is divided into five sections: starting at the top we have the pattern area, then the command words, the brush shapes, the function area and finally the current pattern display. The position of the graphics cursor is shown by either one of the paint brush shapes or by one of the command symbols – pencil, rubber etc – but it changes to a flashing square

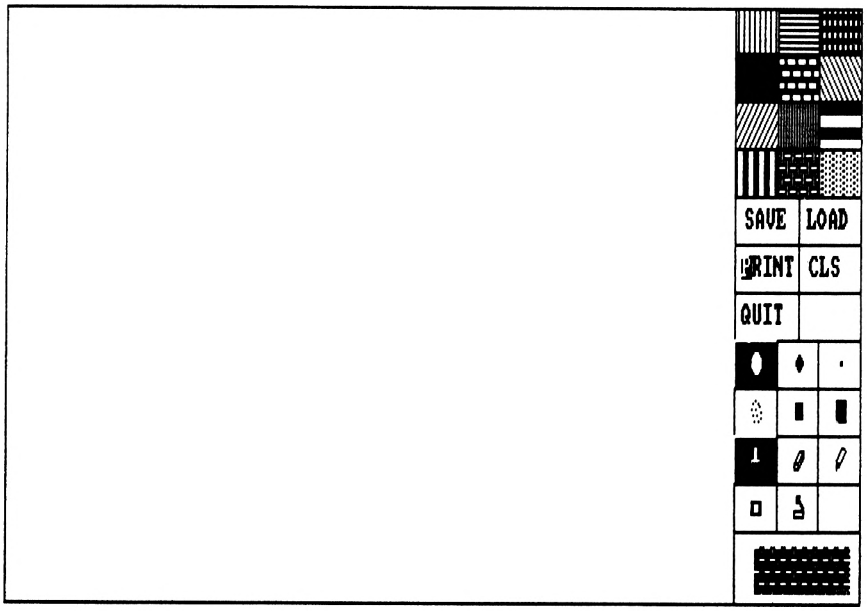


Fig 5.4 The menu displayed by AMART

when it moves into the menu area. At the top of the menu there are a number of patterns displayed, any of which can be selected for use by the paint brush. The current pattern is displayed in a large rectangle at the bottom of the menu. What you can do when the graphics cursor is in the drawing area depends on the function you have selected. The currently selected function or brush shape is marked by being printed in inverse colours.

If you have selected the brush then the graphics cursor appears as the currently selected brush shape in the currently selected pattern. You can move this around the screen without affecting anything that is already present. If you press the COPY key the brush will paint the current pattern as it moves; pressing it a second time returns the brush to its original state. If you select the rubber then you can rub out foreground (black) pixels by moving the rubber over them and pressing COPY. Selecting the pencil will produce a rubber band line on the screen. You can make the displayed line permanent by pressing COPY and you can move the fixed point to the current position of the pencil by pressing F7 (that is 7 on the numeric keypad). Similarly selecting the square will produce a rubber band rectangle, one corner of which is fixed and the other which can be moved using the cursor keys. The rectangle can be made permanent by pressing COPY and the fixed corner can be moved to the current position of the moving corner by pressing F7. Finally if the roller is selected nothing happens until the COPY key is pressed when a scan fill is performed, starting from the position of the roller and using the currently selected pattern. All of the drawing on the screen is performed using an OR graphics ink mode so it has the effect of adding to, rather than overwriting, what is already there.

The command words are self-explanatory and selecting one of them causes the corresponding action to happen. The SAVE and LOAD commands are worth commenting on as they begin working immediately without printing any messages (for obvious reasons) and save and load the screen data using a file called 'PIC'. This use of a single name works well for tape if you restrict yourself to one drawing stored per tape but for disc it is necessary to rename files from 'PIC' to 'PIC1' and so on to avoid confusion.

A feature of the cursor keys is that they can be used in combination to produce diagonal movement. Normally the graphics cursor moves one pixel at a time and this tends to be too slow for most operations apart from accurate drawing. To move the graphics cursor by a larger amount each time, all that is necessary is to press the F9 key (9 on the numeric key pad). A complete summary of the key functions is

cursor keys	–move the graphics cursor and hence the currently selected function icon or the control cursor
COPY	–draw command for functions that draw pixels and an erase command for the eraser
F9	–increase the amount the cursor moves at each step
F7	–move the fixed point in rubber band operations to the current location of the movable point

Note: F9 and F7 are the 9 and 7 keys on the numeric keypad just below the cursor keys.

The complete program listing can be seen below. It is a long listing so take care when you enter it. If a particular function or feature doesn't work when you first try it then make use of the subroutine table given in the next section to narrow down the area in which the typing error is likely to be.

```
10   MODE 2
20   GOSUB 1000
30   GOSUB 5000
40   WHILE QUIT%=0
50    GOSUB 2000
80    GOSUB 500
100  WEND
199  STOP

500  IF X%+DX%>=536 THEN GOSUB 4500:RETURN
510  IF F%=1 THEN GOSUB 4000:RETURN
520  IF F%=2 THEN GOSUB 4800:RETURN
530  IF F%=3 THEN GOSUB 4900:RETURN
540  IF F%=4 THEN GOSUB 4200:RETURN
550  IF F%=5 THEN GOSUB 4600
560  RETURN

1000 REM INIT
1010 INK 0,26
1020 INK 1,0
1030 PRINT CHR$(23);CHR$(0);
1040 P%=3:REM P% IS PATTERN NUMBER
1050 S%=1:REM S% IS BRUSH NUMBER
1060 D%=0:REM D% IS DRAW INDICATOR
1065 F%=1:REM F% IS FUNCTION
1066 M%=0:REM IS HIGH RES MARKER
1070 REM pattern data
1080 DATA &8888,&8888,&8888,&8888,&8888,&8888,&8888,&8888
1090 DATA &8888,&8888,&8888,&8888,&8888,&8888,&8888,&8888
1100 DATA &FFFF,&0000,&FFFF,&0000,&FFFF,&0000,&FFFF,&0000
```

```
1110 DATA &FFFF,&0000,&FFFF,&0000,&FFFF,&0000,&FFFF,&0000
1120 DATA &FFFF,&FFFF,&3333,&3333,&FFFF,&FFFF,&3333,&3333
1130 DATA &FFFF,&FFFF,&3333,&3333,&FFFF,&FFFF,&3333,&3333
1140 DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
1150 DATA &FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF,&FFFF
1160 DATA &FFFF,&8181,&8181,&FFFF,&FFFF,&1818,&1818,&FFFF
1170 DATA &FFFF,&8181,&8181,&FFFF,&FFFF,&1818,&1818,&FFFF
1180 DATA &1111,&2222,&4444,&8888,&1111,&2222,&4444,&8888
1190 DATA &1111,&2222,&4444,&8888,&1111,&2222,&4444,&8888
1200 DATA &8888,&4444,&2222,&1111,&8888,&4444,&2222,&1111
1210 DATA &8888,&4444,&2222,&1111,&8888,&4444,&2222,&1111
1220 DATA &AAAA,&AAAA,&AAAA,&AAAA,&AAAA,&AAAA,&AAAA,&AAAA
1230 DATA &AAAA,&AAAA,&AAAA,&AAAA,&AAAA,&AAAA,&AAAA,&AAAA
1240 DATA &FFFF,&FFFF,&FFFF,&FFFF,&0000,&0000,&0000,&0000
1250 DATA &FFFF,&FFFF,&FFFF,&FFFF,&0000,&0000,&0000,&0000
1260 DATA &F0F0,&F0F0,&F0F0,&F0F0,&F0F0,&F0F0,&F0F0,&F0F0
1270 DATA &F0F0,&F0F0,&F0F0,&F0F0,&F0F0,&F0F0,&F0F0,&F0F0
1280 DATA &AAAA,&FBFB,&0A0A,&FBFB,&AAAA,&BFBF,&A0A0,&BFBF
1290 DATA &AAAA,&FBFB,&0A0A,&FBFB,&AAAA,&BFBF,&A0A0,&BFBF
1300 DATA &4242,&1818,&4242,&1818,&4242,&1818,&4242,&1818
1310 DATA &4242,&1818,&4242,&1818,&4242,&1818,&4242,&1818

1500 DIM C%(15,12)
1510 FOR J%=1 TO 12
1520  FOR I%=0 TO 15
1530   READ C%(I%,J%)
1540  NEXT I%
1550 NEXT J%
1590 REM brush data
1600 DATA &3C,&7E,&FF,&FF,&FF,&FF,&7E,&3C
1610 DATA &00,&18,&3C,&7E,&7E,&3C,&18,&00
1620 DATA &00,&00,&00,&18,&18,&00,&00,&00
1630 DATA &10,&44,&02,&90,&01,&48,&01,&24
1640 DATA &00,&7E,&7E,&7E,&7E,&7E,&7E,&00
1650 DATA &FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
1700 DIM B%(7,6)
1710 FOR J%=1 TO 6
1720  FOR I%=0 TO 7
1730   READ B%(I%,J%)
1740  NEXT I%
1750 NEXT J%
1800 X%=530:Y%=200
1850 DIM T%(15)
1860 T%(0)=1
1870 FOR I%=1 TO 14
1880  T%(I%)=T%(I%-1)*2
1890 NEXT I%
1895 T%(15)=&8000
1900 REM FUNCTION SHAPES
1910 DATA &38,&38,&38,&38,&FE,&AA,&AA,&AA
1920 DATA &00,&0F,&13,&25,&4A,&F4,&98,&F0
```

```
1930 DATA &0E,&11,&22,&44,&88,&F0,&E0,&80
1940 DATA &00,&00,&FF,&C3,&C3,&C3,&FF,&00
1945 DATA &38,&38,&3C,&6,&3,&FF,&81,&FF
1950 DIM F%(7,5)
1960 FOR J%=1 TO 5
1970  FOR I%=0 TO 7
1980   READ F%(I%,J%)
1990  NEXT I%
1995 NEXT J%
1999 RETURN

2000 REM get key
2020 DX%=0:DY%=0
2030 IF INKEY(0)=0 THEN DY%=2
2040 IF INKEY(2)=0 THEN DY%=-2
2050 IF INKEY(1)=0 THEN DX%=1
2060 IF INKEY(8)=0 THEN DX%=-1
2070 IF INKEY(9)=0 THEN D%=NOT D%
2080 IF INKEY(3)=0 THEN DX%=DX%*8:DY%=DY%*8
2090 IF INKEY(10)=0 THEN M%=-1
2999 RETURN

3000 A%=(X% MOD 16)
3010 B%-(Y%2) MOD 16
3020 PRINT CHR$(25);CHR$(240);
3030 FOR I%=7 TO 0 STEP -1
3040  M%=C%((B%+I%) MOD 16,P%)
3050  IF A%>0 THEN GOSUB 3500:REM ROTATE LEFT A% TIMES
3060  M%=M% AND B%(7-I%,S%)
3070  PRINT CHR$(M%);
3080 NEXT I%
3099 RETURN

3500 M$=BIN$(M%,16)
3510 M$="&X"+RIGHT$(M$,16-A%)+LEFT$(M$,A%)
3520 M%=VAL(M$)
3999 RETURN

4000 REM draw char
4010 PRINT CHR$(23);CHR$(1);
4020 TAG
4030 MOVE X%,Y%
4040 PRINT CHR$(240);
4044 TAGOFF
4045 IF D% THEN PRINT CHR$(23);CHR$(3);:MOVE X%,Y%:
     TAG:PRINT CHR$(240);:TAGOFF
4050 X%=X%+DX%
4060 Y%=Y%+DY%
4065 PRINT CHR$(23);CHR$(1);
4070 MOVE X%,Y%
4074 GOSUB 3000
```

```
4075 TAG
4080 PRINT CHR$(240);
4090 TAGOFF
4100 RETURN

4200 REM BOX
4210 PRINT CHR$(23);CHR$(1);
4215 MOVE X%,Y%
4216 TAG:PRINT CHR$(240);:TAGOFF
4217 X%=X%+DX%:Y%=Y%+DY%
4220 MOVE X2%,Y2%:DRAW X%,Y2%:DRAW X%,Y%-16:
     DRAW X2%,Y%-16:DRAW X2%,Y2%
4270 PRINT CHR$(25);CHR$(240);
4280 FOR I%=0 TO 7
4290  PRINT CHR$(F%(I%,F%));
4300 NEXT I%
4310 IF D% THEN PRINT CHR$(23);CHR$(3);:M%=-1
4320 MOVE X2%,Y2%:DRAW X%,Y2%:DRAW X%,Y%-16:
     DRAW X2%,Y%-16:DRAW X2%,Y2%
4330 PRINT CHR$(23);CHR$(1);
4340 MOVE X%,Y%
4345 TAG:PRINT CHR$(240);:TAGOFF
4350 IF M% THEN X2%=X%:Y2%=Y%-16:M%=0
4360 D%=0
4370 RETURN

4500 REM CONTROL
4505 MOVE X%,Y%
4510 TAG:PRINT CHR$(240);:TAGOFF
4520 SYMBOL 240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
4525 PRINT CHR$(23);CHR$(1);
4530 X%=X%+DX%
4540 Y%=Y%+DY%
4545 MOVE X%,Y%
4550 TAG:PRINT CHR$(240);:TAGOFF
4554 D%=0
4555 IF INKEY(9)<>0 THEN RETURN
4560 IF Y%>=280 THEN GOSUB 7000:RETURN
4570 IF Y%>=192 THEN GOSUB 7100:RETURN
4580 IF Y%>=128 THEN GOSUB 6000:RETURN
4590 IF Y%>=64  THEN GOSUB 6100:RETURN
4595 RETURN

4600 REM FILL ROLLER
4610 PRINT CHR$(23);CHR$(1);
4620 MOVE X%,Y%
4630 TAG:PRINT CHR$(240);:TAGOFF
4640 X%=X%+DX%
4650 Y%=Y%+DY%
4655 IF D% THEN GOSUB 7500
4660 PRINT CHR$(25);CHR$(240);
```

```
4670 FOR I%=0 TO 7
4680  PRINT CHR$(F%(I%,F%));
4690 NEXT I%
4700 MOVE X%,Y%
4710 TAG:PRINT CHR$(240);:TAGOFF
4730 D%=0
4740 RETURN

4800 REM RUBBER
4810 PRINT CHR$(23);CHR$(1);
4820 MOVE X%,Y%
4830 TAG:PRINT CHR$(240);:TAGOFF
4840 X%=X%+DX%
4850 Y%=Y%+DY%
4855 PRINT CHR$(25);CHR$(240);
4856 FOR I%=0 TO 7
4857  PRINT CHR$(F%(I%,F%));
4858 NEXT I%
4859 IF D% THEN PRINT CHR$(23);CHR$(0);:TAG:
     MOVE X%,Y%:PRINT CHR$(32);:TAGOFF
4860 PRINT CHR$(23);CHR$(1);
4865 MOVE X%,Y%
4870 TAG:PRINT CHR$(240);:TAGOFF
4880 D%=0
4890 RETURN

4900 REM PENCIL
4910 PRINT CHR$(23);CHR$(1);
4911 MOVE X%,Y%
4912 TAG:PRINT CHR$(240);:TAGOFF
4913 X%=X%+DX%:Y%=Y%+DY%
4915 MOVE X2%,Y2%:DRAW X%,Y%-16
4936 PRINT CHR$(25);CHR$(240);
4937 FOR I%=0 TO 7
4938  PRINT CHR$(F%(I%,F%));
4939 NEXT I%
4940 IF D% THEN PRINT CHR$(23);CHR$(3);:M%=-1
4944 MOVE X2%,Y2%
4945 DRAW X%,Y%-16
4950 MOVE X%,Y%
4960 PRINT CHR$(23);CHR$(1);
4970 TAG:PRINT CHR$(240);:TAGOFF
4980 IF M% THEN X2%=X%:Y2%=Y%-16:M%=0
4985 D%=0
4990 RETURN

5000 REM PRINT SCREEN
5010 CLG
5020 P%=0
5030 S%=6
5110 FOR Q%=1 TO 4
```

```
5120  FOR R%=1 TO 3
5130 U%=65+(INT((Q%-1)/6)+R%)*4
5140 V%=(Q%*2)-1
5150 P%=P%+1
5160 GOSUB 3000
5170 LOCATE U%,V%
5180 PRINT CHR$(240);CHR$(240);CHR$(240);CHR$(240);
5190 LOCATE U%,V%+1
5200 PRINT CHR$(240);CHR$(240);CHR$(240);CHR$(240);
5210  NEXT R%
5220 NEXT Q%
5240 ORIGIN 0,0
5250 MOVE 544,400
5260 DRAWR 0,-400
5270 MOVE 576,400
5280 DRAWR 0,-128
5290 MOVE 607,400
5300 DRAWR 0,-128
5310 FOR Q%=400 TO 32 STEP -32
5320  MOVE 544,Q%
5330  DRAWR 96,0
5340 NEXT Q%
5350 MOVE 592,272
5360 DRAWR 0,-96
5370 MOVE 551,264
5380 TAG:PRINT "SAVE";:TAGOFF
5390 MOVE 598,264
5400 TAG:PRINT "LOAD";:TAGOFF
5410 MOVE 549,232
5420 TAG:PRINT "PRINT";:TAGOFF
5430 MOVE 600,232
5440 TAG:PRINT "CLS";:TAGOFF
5450 MOVE 549,200
5460 TAG:PRINT "QUIT";:TAGOFF
5500 MOVE 576,176
5510 DRAW 576,48
5520 MOVE 607,176
5530 DRAW 607,48
5600 FOR B%=1 TO 6
5610  PRINT CHR$(25);CHR$(240);
5620  FOR I%=0 TO 7
5630   PRINT CHR$(B%(I%,B%));
5640  NEXT I%
5650  IF B%<4 THEN MOVE 525+32*B%,168 ELSE
      MOVE 525+32*(B%-3),136
5670  TAG:PRINT CHR$(240);:TAGOFF
5680 NEXT B%
5685 FOR F%=1 TO 5
5686  PRINT CHR$(25);CHR$(240);
5687  FOR I%=0 TO 7
5688   PRINT CHR$(F%(I%,F%));
```

```
5690  NEXT I%
5700  IF F%<4 THEN MOVE 525+32*F%,104 ELSE
      MOVE 525+32*(F%-3),72
5710  TAG:PRINT CHR$(240);:TAGOFF
5720 NEXT F%
5900 P%=1
5910 GOSUB 8000
5920 PRINT CHR$(23);CHR$(1);
5925 SYMBOL 240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
5930 S%=1
5935 GOSUB 6500
5936 F%=1
5937 GOSUB 6600
5940 GOSUB 3000
5950 MOVE X%,Y%
5960 TAG:PRINT CHR$(240);:TAGOFF
5999 RETURN

6000 REM change brush
6010 GOSUB 6500
6020 S%=(X%+3)32-16+((407-Y%)32-7)*3
6030 GOSUB 6500
6040 RETURN

6100 REM CHANGE FUNCTION
6110 GOSUB 6600
6120 F%=(X%+3)32-16+((407-Y%)32-9)*3
6130 GOSUB 6600
6140 RETURN

6500 REM MARK SELECTED BRUSH
6510 IF S%<4 THEN MOVE 512+32*S%,176 ELSE
     MOVE 512+32*(S%-3),144
6520 TAG:PRINT STRING$(4,CHR$(240));:TAGOFF
6530 IF S%<4 THEN MOVE 512+32*S%,160 ELSE
     MOVE 512+32*(S%-3),128
6540 TAG:PRINT STRING$(4,CHR$(240));:TAGOFF
6550 RETURN

6600 REM MARK SELECTED FUNCTION
6610 IF F%<4 THEN MOVE 512+32*F%,112 ELSE
     MOVE 512+32*(F%-3),80
6620 TAG:PRINT STRING$(4,CHR$(240));:TAGOFF
6630 IF F%<4 THEN MOVE 512+32*F%,96 ELSE
     MOVE 512+32*(F%-3),64
6640 TAG:PRINT STRING$(4,CHR$(240));:TAGOFF
6650 RETURN

7000 REM change pattern
7010 GOSUB 8000
7020 P%=(X%+3)32-16+((407-Y%)32)*3
```

```
7030 GOSUB 8000
7040 SYMBOL 240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
7050 RETURN

7100 REM COMMANDS
7110 C%=(636-X%)48+((407-Y%)32-4)*2
7120 IF C%=0 THEN GOSUB 9000:REM LOAD
7130 IF C%=1 THEN GOSUB 9100:REM SAVE
7140 IF C%=2 THEN GOSUB 9200:REM CLS
7150 IF C%=3 THEN GOSUB 9300:REM PRINT
7160 IF C%=5 THEN GOSUB 9400:REM QUIT
7170 RETURN

7500 REM FILL AREA WITH CURRENT PATTERN
7505 X%=X%+4:Y%=Y%-16
7510 XT%=X%:YT%=Y%
7520 WHILE YT%>=0 AND TEST(XT%,YT%)=0
7530  GOSUB 7900
7540  XT%=XT%-1
7550  GOSUB 7800
7560  YT%=YT%-2:XT%=X%
7570 WEND
7580 XT%=X%:YT%=Y%+2
7590 WHILE YT%<=399 AND TEST(XT%,YT%)=0
7600  GOSUB 7900
7610  XT%=XT%-1
7620  GOSUB 7800
7630  YT%=YT%+2:XT%=X%
7640 WEND
7645 X%=X%-4:Y%=Y%+16
7650 RETURN

7800 REM SCAN LEFT AND FILL
7810 WHILE XT%>=0 AND TEST(XT%,YT%)=0
7820  A%=XT% MOD 16:B%=(YT%2+1) MOD 16
7830  IF C%(B%,P%) AND T%(15-A%) THEN PLOT XT%,YT%,1
7840  XT%=XT%-1
7850 WEND
7860 RETURN

7900 REM SCAN RIGHT
7910 WHILE XT%<536 AND TEST(XT%,YT%)=0
7920  XT%=XT%+1
7930 WEND
7999 RETURN

8000 REM current pattern
8005 XT%=X%:YT%=Y%:ST%=S%:S%=6
8010 FOR X%=560 TO 631 STEP 8
8020  FOR Y%=23 TO 40 STEP 16
8030   MOVE X%,Y%
```

```
8035   GOSUB 3000
8040   TAG:PRINT CHR$(240);:TAGOFF
8050  NEXT Y%
8060 NEXT X%
8070 MOVE 544,48
8080 DRAWR 0,-48
8090 X%=XT%:Y%=YT%:S%=ST%
8100 RETURN

9000 LOAD "!PIC"
9010 GOSUB 8000
9015 MOVE X%,Y%
9020 SYMBOL 240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
9030 TAG:PRINT CHR$(240);:TAGOFF
9040 GOSUB 6500
9050 GOSUB 6600
9060 RETURN

9100 SPEED WRITE 1
9110 MOVE X%,Y%
9120 TAG:PRINT CHR$(240);:TAGOFF
9125 GOSUB 6500
9126 GOSUB 6600
9127 GOSUB 8000
9130 SAVE "!PIC",B,49152,16384
9135 GOSUB 8000
9140 MOVE X%,Y%
9145 SYMBOL 240,&FF,&FF,&FF,&FF,&FF,&FF,&FF,&FF
9150 TAG:PRINT CHR$(240);:TAGOFF
9160 GOSUB 6500
9170 GOSUB 6600
9180 RETURN

9200 PRINT CHR$(23);CHR$(0);
9205 LOCATE 1,1
9210 PEN 0
9220 FOR I%=1 TO 25
9230  PRINT STRING$(68,CHR$(143));
9240 NEXT I%
9250 PEN 1
9260 PRINT CHR$(23);CHR$(1);
9270 RETURN

9300 REM PRINTER DUMP
9310 REM ******************************************
9320 REM INSERT YOUR OWN PRINTER DUMP ROUTINE HERE
9330 REM ******************************************
9340 RETURN

9400 QUIT%=-1
9410 RETURN
```

How Amart works

If you are only interested in using AMART as an example of the techniques described in the previous section then you can feel free to skip this explanation of its inner workings. As AMART is such a long program there is no way of giving a full explanation in a reasonable amount of space, and indeed much of it is obvious enough not to need explanation. An overall view of the structure of the program can be seen from its subroutine table given below.

Subroutine Use – AMART

Subroutine	function
500	Selects subroutine to implement currently selected function
1000	Initialisation – sets up arrays, containing data that define patterns, brush shapes and icons
2000	Reads cursor keys, COPY, F9 and F7 and sets variables accordingly – replace this routine if you want to use another input device
3000	Defines character 240 to be current pattern appropriate for the present position of the graphics cursor
3500	Shifts bit pattern in M% A% bits to the left
4000	Implements paint brush
4200	Implements rubber band rectangle
4500	Control routine – if graphics cursor is in the menu area this subroutine prints the control cursor and tests for the COPY key. If the COPY key is pressed it calls an appropriate subroutine to implement the selection
4600	Scan fill routine – if COPY key has been pressed it calls subroutine 7500 to do the fill
4800	Erase routine – uses the eraser character to set pixels to the background colour if the COPY key is pressed
4900	Implements rubber band pencil
5000	Prints initial screen including menu
6000	Changes brush – S% is the brush number
6100	Changes function – F% is function number
6500	Changes the selected brush's menu box to inverse colours
6600	Changes the selected function's menu box to inverse colours
7000	Changes current pattern – P% is pattern number
7100	Calls subroutine to carry out one of the 'word' commands – SAVE, LOAD etc

7500	Scan fills an area with the current pattern
7800	Scans left and fills
7900	Scans right
8000	Prints current pattern at bottom of menu column
9000	Loads picture
9100	Saves picture
9200	Clears the screen
9300	Printer dump – insert your own printer dump subroutine or machine code call here
9400	Quits program

The main program takes the form of a WHILE loop that repeatedly calls subroutine 1000 to update the cursor position and read the other keys, and subroutine 500 to implement the currently selected function. Most of the subroutines work in a fairly obvious way if you consider what they have to do and recall the explanations given earlier. However there are one or two features of the program that deserve further comment.

One of the main features is the provision of a range of patterns. Each pattern is defined in squares 16 pixels by 16 pixels. The DATA statements, line 1080 to 1310 define the twelve different patterns shown in the menu. Each pair of DATA statements defines 16 rows of 16 dots in much the same way that a SYMBOL statement defines 8 rows of eight dots for a standard size character. So for example DATA statements 1080 and 1090 define the first pattern in the top left of the menu. All of the pattern data is read into the array C%, in such a way that C%(R%,P%) contains the definition of row R% for pattern P%. The brush shapes are defined in the same way but only as 8 by 8 pixels in size by DATA statements 1600 to 1650. The brush definitions are stored in the array B% in such a way that B%(R%,S%) contains the definition of row R% for brush S%. Finally, the icons used for the functions are defined as 8 by 8 characters by DATA statements 1910 to 1945. The icon definitions are stored in the array F% in such a way that F%(R%,F%) contains the definition of row R% for icon F%.

The way that the pattern brush is implemented is a little difficult to explain as it involves bit manipulation, but essentially it works by taking account of the symmetry inherent in the pattern. The fundamental idea is that before the brush character, CHR$(240), is printed subroutine 3000 is called to set up its definition depending on the current cursor position, the current pattern and the currently selected brush shape. It does this by ANDing each row of the brush definition with an appropriate row from the pattern definition. As the pattern repeats in square 16 by

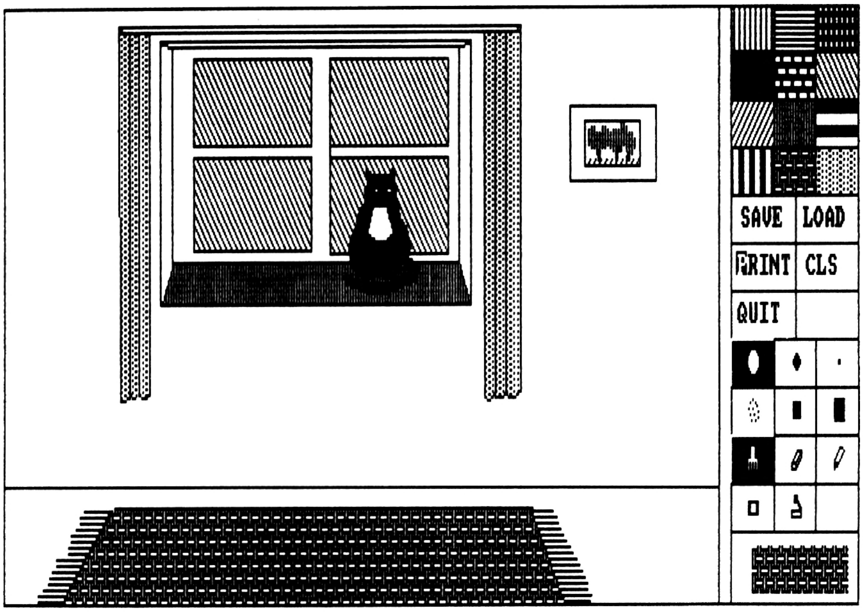


Fig. 5.5 AMART – sample picture 1

16 pixel blocks, the brush is either completely within one of these block or it overlaps as many as four of them. If the brush is completely within a block then the pattern that it should produce on the screen can be found from the block as it is defined by the DATA statements in subroutine 1000. If the brush overlaps more than one block then the pattern defined by the DATA statements has to be shifted, horizontally and vertically, to give the pattern to be found in a 16 by 16 block that does completely contain the brush. Subroutine 3500 is interesting in that it performs a left rotation of the bit pattern stored in M% by A% bits. This is achieved by converting the number in M% to a string of ones and zeros using BIN$ and then uses string operations to implement the rotate.

The only other routine that uses the pattern definition is the scan fill subroutine at 7500. This follows the method used by the scan fill routine introduced earlier, with the exception of lines 7820 and 7830. Line 7820 works out the row of the pattern and the bit within the pattern corresponding to the point that is about to be plotted – if this is a 1 then the point should be plotted; if it is a 0 the point should not be plotted. The only difficult part is finding out quickly the state of a given bit in a number. This is achieved using a set of binary masks stored in the array T%. Subroutine 1000 initialises T% so that the value in T%(I%) is 2^I%, so

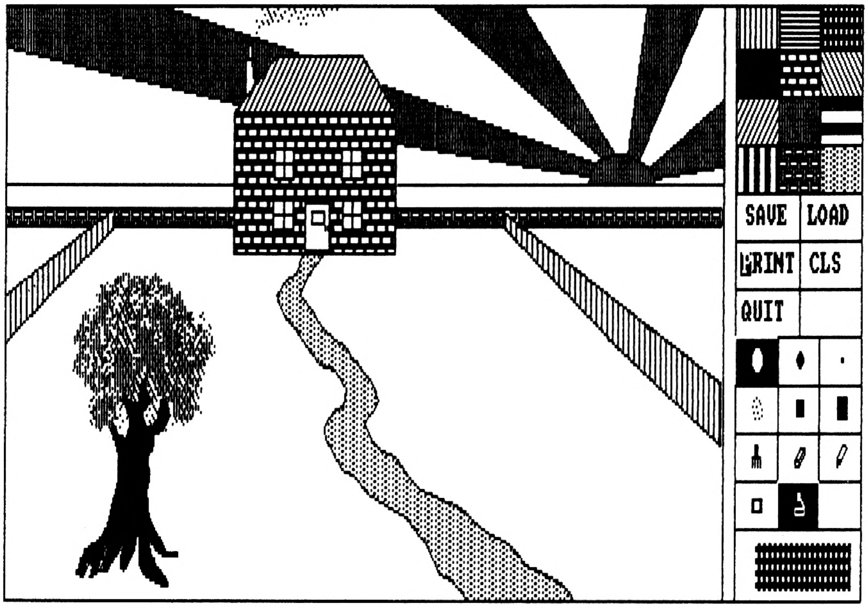


Fig 5.6 AMART – sample picture 2

T%(0)=1, T%(1)=2, T%(2)=4 and so on. The result of ANDing a value with T%(I%) is equal to the Ith bit in the value. For example 5 AND T%(0) is 1 and 5 AND T%(1) is 0 (5=101 in binary).

There are numerous other points of special interest, far too many to go into in detail, but notice the way that the menu selection subroutine use a calculation to find out in which menu box the graphics cursor is positioned (lines 6020, 6120 and 7020). The most obvious way of doing this job is with a long list of IF statements, each one testing for the graphics cursor being positioned over a particular box – the equation method is much easier!

There are many additions and improvements that can be made to AMART and one way to discover how any program works is to modify it. For example you might like to add a rubber band circle drawing routine and a text entry routine so that you can label your drawings. Whether or not you modify it, however, you can have hours of enjoyment drawing pictures with it. The two examples shown in figs 5.5 and 5.6 should give you an idea of the results that can be achieved.

A graphics editor 6

Painting directly on the screen as described in Chapter Five is great fun, but it is very difficult to draw anything exactly. If you want to draw something like a house floor plan or an electronic circuit diagram then 'painting' isn't really the right approach. What is needed is some method that gives more accurate control over where to place objects on the screen.

In the same way that the painting program given in the previous chapter was developed by considering ways of computerising the traditional artistic methods it is possible to develop a computerised 'technical drawing board'. Although technical drawing sounds very dull and uninteresting it is the only sensible approach to producing an accurate representation of something. As you might expect, producing a software technical drawing board uses co-ordinate based graphics to enable accurate positioning of objects and so this chapter could just as easily have been called 'co-ordinate based graphics'. Some co-ordinate based methods have been introduced in Chapter Four and in this chapter these are extended to include everything needed to understand and use the 'graphics editor' program given at the end of the chapter.

Points, lines and polygons

The fundamental graphics entity is the point or pixel but, while any shape can be considered as a collection of points, this is not a practical way for a user to enter or modify a shape. It is fairly obvious that any curve can be approximated by straight line segments, the accuracy of the approximation increasing with the number of lines used, and this suggests that a graphics package based on nothing but straight line manipulation would be almost as good as one that allowed the use of arbitrary curves. Once you decide to ignore general curves and use nothing but straight lines there emerges a simple hierarchy of 'graphics objects'. The point is, of course, still fundamental but the straight line is the element that is used to construct closed shapes or polygons. In the same way that a straight line is defined by the position of two points, a polygon is defined by the position of a number of lines.

There are times when a graphics package should allow the user to interact with individual points, individual lines or collection of lines. For example, if you want to move the position of one end of a line then it is natural to select the point that defines the end of the line and move it. On the other hand if you want to move the entire line then having to move each end point in turn many be a little awkward!

Point and line files

Making up shapes out of a collection of lines raises the issue of how the data that defines each line's position, orientation and length should be stored. You could, as many graphics programmers do, choose to ignore the problem and simply store the information in any old way that occurs to you as you write the program. However there is a great deal to be gained from thinking out a clear and structured way of storing line data so that it can be easily modified. The obvious way to store a collection of points is as a pair of arrays X%(I) and Y%(I) used to hold the x and y co-ordinates of each point. This method of holding points is known as a 'point file' and it has the advantage that the index of the arrays also serves to identify each point and thus makes editing possible.

The most obvious way to define a line segment is to give the location of its starting point and its end point. You might think that this implies that the best way to store line information is in four arrays used to store the two pairs of co-ordinates of the starting and finishing points respectively. However, as lines that make up a single shape tend to share start and endpoints it is better to store the co-ordinates of the points in a point file and use the point indices to specify pairs of points. For example, a square would be stored as

X%(I)	Y%(I)	S%(I)	E%(I)
0	0	1	2
1	0	2	3
1	1	3	4
0	1	4	1

The two arrays X%(I) and Y%(I) store the co-ordinates of four points 0,0 1,0 1,1 and 0,1, which are the four corners of a square. The two arrays S%(I) and E%(I) store the start and end points of four lines. For example, the first line starts at point 1 and ends at point 2. In other words, the line is between X%(S%(1)), Y%(S%(1)) and X%(E%(1)), Y%(E%(1)). In general, the Ith line is drawn between X%(S%(I)), Y%(S%(I)) and X%(E%(I)), Y%(E%(I)).

This is not the only way to store a list of lines that make up a display but it has many advantages that become clear only when you try to change the arrangement of lines, so as to make a shape bigger or change its orientation.

Entering graphics data

Now that the method of storing point and line information has been decided it is easy to see how, given the arrays X%,Y%,S% and E%, to draw the scene that they describe. What is not so easy is to suggest a suitable way of getting the information into the arrays in the first place or how to change the information interactively. As in the case of computer-assisted art, a technical graphics system should be as easy to use as the traditional technology of pencil, paper, drawing board and set square. In fact to be worthy of replacing the traditional materials it has to be in some way better. Of course even a badly designed graphics system offers certain advantages over pencil and paper just because a computer is involved, but this is not a reason for avoiding the problems of designing a good input and editing method.

The method used to input graphics data is obviously dependent on the hardware available. Technical graphics makes the same demands on an input device that artistic graphics makes. It must be quick, easy and natural to use, but in addition it must be accurate. A technical graphics input device must allow the user to set the position of a point to within one pixel without any difficulty. Most of the input devices discussed in the previous chapter, the light pen, the joystick and the mouse, are also suitable for technical graphics if they are of a sufficiently high quality. However, such devices are always used to control the position of a graphics cursor, and so, to avoid the need for special hardware, the program developed in this chapter uses the cursor keys in much the same way as the painting program in the previous chapter.

Once again keyboard positioning has to be both accurate and fast and the best solution to this problem has already been described – the addition of a fifth key (F9 or 9 on the numeric keypad) to the usual four cursor keys to control the speed of movement. Pressing one of the directions keys on their own should move the cursor by the smallest increment that the graphics display can resolve – so allowing accurate positioning, but pressing one of the directions keys at the same time as the 'speed' key should move the cursor by a much larger increment. As the program is intended to be used for accurate input the choice of increments is important. For example even though changing the Y co-ordinate by one doesn't always move the graphic cursor on to a new pixel it is often necessary to input graphics data more accurately than it can be displayed. For this reason the graphics editor program allows the user to move the graphics cursor by one unit in both the X and Y directions by pressing the appropriate cursor keys. The 'speed up' factor is chosen to produce an increment of 10 units in both directions. As the current X and Y co-ordinates of the graphics cursor are displayed at the bottom right hand side of the screen (see fig 6.1) this makes setting an exact position easy. Notice that using this system more than one arrow key can be used at a time. For example to move the cursor diagonally up the screen and to the left at great speed all you have to do is press the up and left arrow key while holding down the F9 key.

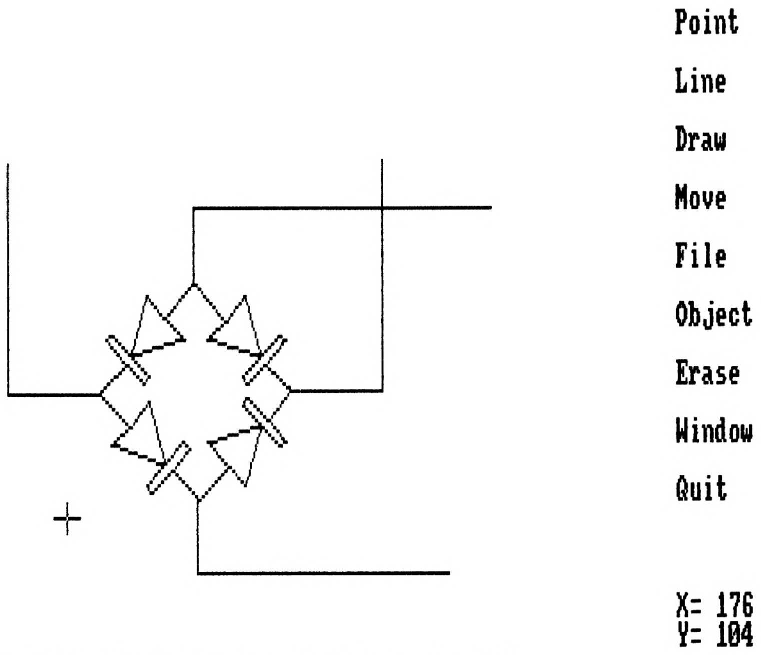


Fig 6.1 Bridge rectifier circuit drawn using AMPLOT

Using the keyboard to control the position of a graphics cursor it is now easy to see how to input single points. The graphics cursor can be moved to the desired location and then pressing another key, 'P' say, its current co-ordinates would be stored in the appropriate location in the point file, that is the arrays X% and Y%. If P% is used to record the number of points already entered and the graphics cursor position is stored in XG% and YG% then entering a point is simply

```
P%=P%+1
X%(P%)=XG%
Y%(P%)=YG%
PLOT XG%,YG%
```

where the plot is included to show the user that the point has indeed been added to the drawing.

The next problem is to find a way of storing line data in S% and E%. The restriction that lines can only be specified between points that have already been defined makes this task much easier. A point already on the screen can be specified using 'cyclic selection'. When the user wants to input a line the graphics cursor can be automatically positioned at one of the points on the screen. If this is one of the two points that determine the line then the user can simply press a key to indicate that this is the case. On the other hand, if it is not, pressing another key makes the graphics cursor move on to the next point. In this way the user can make the graphics cursor 'cycle' through all of the points on the screen until the one required is located. Cyclic selection does have a number of disadvantages which will be discussed later but it is simple and surprisingly pleasant to use. When a point has been selected as the start of a line then its number is stored in S%. Similarly if it is selected as an endpoint its number is stored in E%.

Editing graphics data – rubber banding

Now that a method for entering both points and lines has been described the next problem is editing. Removing existing points and lines is not difficult and involves no new ideas, however it is best dealt with later in this chapter, after the introduction of 'objects'. The most important advantage of computer graphics is that it gives (or should give) the user the opportunity to modify a drawing by moving points interactively. For example if you have drawn a square by defining four points and then joining them by four lines it should be possible to 'move' two of the points to turn it into a rectangle. This sort of editing is easy enough to implement. The point to be moved can be determined by cyclic selection and then moved using the arrow keys in the usual way.

The straightforward method of moving a point with the graphics cursor gives very little feedback to the user about how the modified drawing looks. The obvious solution to this problem is to redraw all of the lines to the point being moved, each time it is moved. This gives the user the impression that the lines connected to the point are in some sense 'elastic' and can be stretched or shortened at will. You should be able to recognise this as another application of the 'rubber banding' method introduced in Chapter Five. However this form of rubber banding is a little more complicated in that the number of lines connected to the moving point is variable.

Implementing a rubber banding point move option in the graphics editor is a little more complicated than you might expect, if a reasonable response time is to be maintained. The most obvious method of redrawing the lines connected to a point is to redraw all of the lines that make up the display. This is indeed easy but it is not difficult to appreciate that it quickly becomes unacceptably slow as the number of lines increases. Redrawing only the lines that are connected to the point being moved is clearly the correct way to implement rubber banding. However this involves scanning through the line endpoint arrays S% and E% to find out which lines are connected to the point in question and this takes time. To avoid slowing things down, a list of lines connected to the point has to be constructed just once and then re-used each time the point is moved. The best time to construct this list is when the point is being selected. That is, the cyclic selection routine should return a list of lines connected to the point that is being selected. The list can then be used by the point move routine to erase and redraw all of the lines each time the point is moved.

Objects

Initially, it is sufficient to enter and manipulate individual points and lines, but as the complexity of a drawing increases it becomes desirable to work in terms of discrete collections of lines – that is 'objects'. For example, if you are building up a simple circuit diagram such as that shown in fig 6.2, at first you will want to manipulate individual points to alter the shape of the triangular elements but later it becomes a difficult chore to have to move an entire triangle by moving each of its points independently. What is required is some method of defining 'objects', as collections of particular lines, and performing operations on these objects.

The most logical way of adding objects and object operations to a graphics editor is to introduce an 'object file' that lists the line numbers of the lines that make up each object. That is, each element of the object file would be a list of all the line numbers that define the object. Unfortunately, unlike the point and line files already described, each element of an object file would have to be capable of storing a variable number of line numbers. For example, object number 1 might be a triangle which needs three line numbers stored, but object number 2 might be a square needing four line numbers to define it. BASIC doesn't really provide any simple way of constructing such variable length lists – it can be done, but not easily. A suitable compromise is to use an array O% to store the object number of each line. That is line I% belongs to object number O%(I%). This restricts each line to belonging to just one object but this proves not to be too important in practice. Using this system, drawing a given object is simply a matter of scanning through the entire line file and drawing only the lines with the appropriate object number. That is to draw object OB%

```
FOR I%=1 TO L%
IF O%(I%)=OB% THEN draw line I%
NEXT I%
```

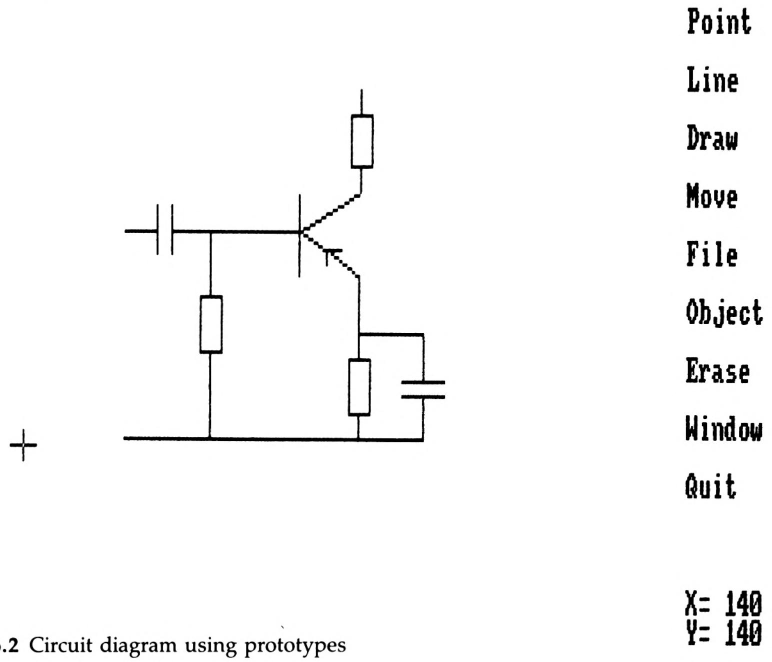


Fig 6.2 Circuit diagram using prototypes

However, as in the case of rubber banding, it is better to construct a list of the lines that make up the object and then use this list to draw the lines.

In AMPLOT, the graphics editor program given later, objects are identified by numbers starting from 0. Each line in the line file can belong to one and only one object. Although points are not assigned to objects it is useful to consider a point as belonging to an object if it is one of the endpoints of a line that belongs to the object. One problem with this simple definition is that, as a point can be an endpoint of a number of lines, it can also belong to more than one object! However, for the moment this complication will be ignored by supposing that all lines with the same endpoint belong to the same object. When lines are entered, the object number that they are assigned depends on where they are drawn. If a line is drawn between two points that 'belong' to an existing object then the line is given the same object number – that is it is appended to that object. If the line is drawn between two points that belong to different objects then it is assigned to object number 0. Object number 0 is very special in that when a drawing is first started there are no other objects defined and so all the lines that are entered are automatically assigned to object zero. Other objects can be created by using the 'object create' option which allows the user to indicate, using cyclic selection, the lines in object 0 that are to be grouped together under a new object number. These new object numbers are assigned sequentially starting with object 1.

These rules for creating objects may sound complicated but they do provide a natural way of working. Initially, lines that are entered are assigned to object 0 which can be thought of as a 'background' object, containing not only lines that are destined to become new objects but also lines that connect points in different objects. New objects are created by selecting lines from object 0, because lines that are already assigned another object cannot be reassigned. (It would be possible to include an option for re-assigning lines but in practice it is not very useful). After an object has been defined, the only way that new lines can be added to it is by drawing between points that belong to it.

Object cycling and erasing lines

The idea of cycling through all of the points in the point file was introduced earlier as a simple way of allowing the user to select an existing position within the drawing. In the same way you can use line cycling to determine which line will be involved in a line operation and object cycling to determine which object will be involved in an object operation. However, if for example the point file reaches any appreciable size, simple point cycling quickly becomes very tedious because of the number of points that have to be cycled though to read the one you want. The best approach to selection is to use hierarchical cycling or simply 'object cycling' for every selection task. For example, to select a line the program should allow the user to cycle around from object to object until the object that contains the desired line is reached. Then the program should allow the user to cycle around the lines within the object to determine exactly which line is required. The same principle applies to selecting a point – first select an object, then the point within the object. In other words, all cycling should be within the current object and there should be an additional command to change the current object.

Object cycling brings one difficulty with it. All the lines in the line file are assigned to an object and so it is possible to reach any line by first cycling through objects and then through lines that belong to that object. However this is not the case for all points because it is possible to have a point that is not an endpoint for any line and so not part of any object. To solve this problem the definition of object 0 has to be extended to include all isolated points. Thus cycling through the points in object zero provides access to points that would otherwise be inaccessible.

As an example of object selection and the general difficulties in doing the book keeping necessary to keep track of object definitions, consider the problem of deleting a line from a drawing. Object cycling should first be used to locate the required line. Deleting the line from the line file is quite easy. For example, if you want to delete line D% all you have to do is move all the lines below it in the line file up by one place and subtract one from L%. To keep the object file correct, it too has to be 'moved up' in the same way. There is one subtle complication of deleting lines in this way that has to be taken into account. If all of the lines that go to a particular point are deleted then it makes sense to delete that point from the line file as well. However deleting a point from the line file is much more complicated than you might think. As well as moving all the points below the one to be deleted up by one place, it also involves correcting the line file for the changes in the point numbers! All-in-all deleting lines is a complex process!

Dragging

Objects only become useful once suitable object operations have been defined. At this stage there is only one obvious object operation – 'dragging'. Dragging is the object equivalent of rubber banding. An object is selected, using cyclic selection between

objects, and then moved as an entire entity using the cursor keys.
Implementing object dragging is nothing more than a matter of keeping track of all the lines and points that make up the object that has been selected. Notice that it is necessary not only to keep a list of the lines that make up an object so that it can be drawn, it is also necessary to keep a list of all the points that are used to define the lines so that their positions can be changed as the object is dragged.

Object transformations

As a drawing is built up the emphasis shifts from the more fundamental points and lines to the manipulation of objects. The reason for this is simply that it is easier to construct an image using a set of standard objects rather than draw everything using points and lines. Ideally the construction of an image should involve nothing but the manipulation of standard, predefined objects. To be able to do this we need ways of changing the position, orientation and size of shapes so that they can be combined to make a complete display.

Changes in position, orientation and size are usually referred to as 'transformations'. For example, a landscape might use a number of tree shapes, identical apart from position, size and orientation, and rather than store the line information for each it is better to keep one 'prototype' shape and generate all of the 'examples' of it using appropriate transformations.

One of the many advantages of storing point and line data separately is that a shape can be transformed by applying the transformation to each of its points in turn and then drawing the straight lines that connect them. For example, if a square is defined by four points and the lines that connect them, a transformation that shifts the four points an equal amount to the left will be effective in shifting the entire square if the lines are redrawn between the same points in their new position. This simple observation means that we need only involve the point file in transformations. Notice that this simplification is only true if the points are connected by straight lines – the transformation of other connecting curves involves applying the transformation to each point on the curve, not just its endpoints.

The next question is what sort of transformations on points are we interested in? If we restrict ourselves to transformations that preserve shape (on the grounds that shape is something defined before the transformation part of a graphics program) then the only transformations permitted (i.e. shape preserving) are translation

rotation

reflection

and scaling (change of size)

As reflection is not something that we are generally interested in this leaves only translation, scaling and rotation to be considered. Each of these transformations can be produced using simple equations tht convert a point's old co-ordinates to a new pair of co-ordinates. For example to translate a point by an amount a in the x direction and b in the y direction the equations

$x' = x + a$
$y' = y + b$

can be used, giving the new position of the point x',y' in terms of its original position x,y. To move an entire object all you do is apply the pair of transformation equations to every point in the point file that is part of the object and then redraw the object's lines. Of course this is exactly what happens during an object drag – see the previous section. For translation, the pair of equations defining the transformation are both simple and obvious and this might make you think that all this theory is unnecessary, but when it comes to scaling and rotation things are more complicated and here the theory is a very definite help.

The equations for the scaling transformation looks deceptively simple

$x' = y\ sx$
$y' = y\ sy$

where sx and sy are 'scaling factors' that govern how much the object will grow in the x and y directions respectively. If this pair of equations is applied to each point that makes up an object then the object will appear to have been stretched by a factor sx in width and a factor sy in height. Normally we are interested in keeping the shape of an object constant and to ensure this sx and sy must be identical. If this is the case then the scaling transformation can be used to adjust the size of an object.

A rotation is the most complicated of the transformations but we have already met the ideas needed in Chapter Four, when drawing a circle. First, it is important to notice that there are two parameters involved in a rotation, the angle through which the object is rotated and the point about which it is rotated. To produce a rotation of an object about the origin each of the points that make up the object have to move along a circular path centred on the origin and with a radius equal to their distance from the origin. Using trigonometry we get

$$x' = x \cos(t) - y \sin(t)$$
$$y' = x \sin(t) + y \cos(t)$$

as the equations that produce a rotation through an angle t about the origin. Applying these equations to each point that makes up an object will cause the entire object to move around the origin through an angle t. Although this is indeed a rotation, it isn't really the sort of rotation that we want. Ideally, we would like the object to stay in roughly the same place while it rotates, so that its orientation could be changed without affecting its position.

In the same way as the rotation transformation changes both the position and orientation of an object, the scaling transformation also has a 'side effect'. Not only does it alter the size of an object it also shifts its position. This is because multiplying co-ordinates in this way moves every point further away from the origin, that is the point 0,0. In fact the scaling transformation described above is better named 'scaling about the origin' in the same way that the rotation transformation is 'rotation about the origin'. So, although we have equations that produce rotation and scaling, neither transformation is exactly what we need. To produce scaling and rotation without shifting the overall position of the object we have to examine the way that transformations can be combined.

Combining transformations – the centroid

The transformation equations given above are very useful, but normally the required transformation is a combination of rotation, translation and scaling. For example, a square in the middle of the screen can be rotated about one of its corners by the following three steps:

1 translating the desired corner to the origin
2 performing the rotation about the origin
3 performing the translation in step 1 in reverse

In the same way, a rotation about any point can be achieved by first translating the point to the origin, performing a rotation about the origin and then performing the reverse translation.

Applying each of the transformation equations in turn can be shown by simple algebra to be equivalent to applying the single transformation

$x' = (x-a)\cos(t) + (y-b)\sin(t) + a$
$y' = (x-a)\sin(t) + (y-b)\cos(t) + b$

which are the transformation equations for rotating an object through an angle t about the point a,b.

Using the same reasoning a scaling about any point can be produced by first applying a translation to bring the point to the origin, then a scaling about the origin and finally a reverse translation. Applying each of these transformations in turn is equivalent to the single transformation

$x' = sx(x-a) + a$
$y' = sy(y-b) + b$

Now that we can both rotate and scale an object about any point, the question is can a rotation or scaling point be found that will result in the object staying in place while it rotates or changes its size? The answer is yes and the point is called the 'centroid'. The centroid is nothing more than the two-dimensional equivalent of the familiar 'centre of gravity'. The co-ordinates of the centroid are simply the average of the x co-ordinates of the object. So whenever a scaling or rotation have to be performed on an object the centroid is calculated and used in the transformation equations given above. Using this scheme objects can be dragged, scaled and rotated with the minimum of interactions between the three activities.

Prototypes

Using the ability to drag, scale and rotate any object it is surprising how easy image construction becomes. For example, the amplifier circuit in fig 6.2 involved defining only three shapes, the transistor symbol, the capacitor symbol and the resistor symbol and transforming them into position. In fact, the best way to use the graphics editor is to first define a range of basic shapes, to form a library for later use.

To make this possible it is, of course, necessary to implement the saving, loading and appending of drawings. If you are using tape then this facility will not be very easy to use but if you are using disc then it is almost as good as a memory resident library of prototypes. For example, if you are going to use the graphics editor to draw electronic circuits all you have to do is to define each of the symbols that you need and store each one in a different file – resistor, capacitor, diode, transistor etc. Then you can draw your circuit simply by appending each symbol that you need and transforming it into position.

Loading and saving are simple enough to implement but appending objects is a little tricky. The basic idea is that appended objects are entered into the object file with an object number that is equal to their original number plus the largest object number so far used in the drawing to which they are being appended. For example, if you save a single object in a file as object number 1 and then append it to a drawing with four objects already defined, it will be entered as object number 5. An object that was saved as object two would be appended to the same drawing as object six.

This is fairly straightforward and trouble-free as long as objects that are saved as object zero are treated differently. For example if object zero were to be appended to a drawing containing four objects according to the standard rules it would be assigned to object four and hence attached to the existing object four. The only sensible thing to do is to treat object zero as special and append it to any existing object zero. So to summarise: when appending objects, the object number to which they are assigned is given by the sum of their original object number plus the largest object number used in the drawing so far – apart from an object saved as object number 0 which is appended to any existing object zero. These rules may sound a little complicated but if you arrange always to save prototypes with an object number of 1 you can virtually forget them. The standard sequence of operations is then:

1 Use the graphics editor to draw any prototype shape that is needed. Define it as object number 1 and store it in prototype library.
2 Recall each prototype that is needed and transform it to the correct position and orientation within the display.
3 Finally, use the graphics editor on the nearly completed display to draw in connecting lines etc that are not part of any prototype. Also at this stage the editor can be used to remove unwanted lines and generally tidy up the final display.

The graphics editor – AMPLOT

A complete listing of AMPLOT, the graphics editor, is given on the pages following.

```
10   REM graphics editor
20   MODE 2
30   GOSUB 1000
40   WHILE QUIT%=0
50    GOSUB 3000
60    GOSUB 4000
70    GOSUB 2500
80    GOSUB 3000
90    XG%=X%:YG%=Y%
100  WEND
199  END

1000 REM init
1010 XG%=100:YG%=100
1020 GOSUB 1200
1030 GOSUB 1300
1040 DIM X%(100),Y%(100)
1050 DIM S%(100),E%(100)
1060 DIM T%(100),O%(100),Q%(100),C%(100)
1070 P%=0
1080 C%=1
1090 X%=XG%
1100 Y%=YG%
1110 L%=0
1120 OB%=0:S%=1:OM%=1
1199 RETURN

1200 REM windows
1210 WINDOW#1,70,80,1,24
1299 RETURN

1300 REM menu1
1320 DATA "Point","Line","Draw"
1330 DATA "Move","File","Object"
1340 DATA "Erase","Window","Quit"
1350 RESTORE 1300
1360 D%=9
1370 GOSUB 2000
1399 RETURN

1400 REM menu2
1420 DATA "Next obj.","Cycle","First","Second"
1430 D%=4
1440 RESTORE 1400
1450 GOSUB 2000
1499 RETURN

1500 REM menu3
1520 DATA "Cycle","Exit"
1530 D%=2
1540 RESTORE 1500
```

```
1550 GOSUB 2000
1599 RETURN

1600 REM menu4
1620 DATA "Save","Load","Append"
1630 D%=3
1640 RESTORE 1600
1650 GOSUB 2000
1699 RETURN

1700 REM menu5
1720 DATA "Create","Drag","Transform"
1730 D%=3
1740 RESTORE 1700
1750 GOSUB 2000
1799 RETURN

1800 REM menu6
1820 DATA "Cycle","Add","Exit"
1830 D%=3
1840 RESTORE 1800
1850 GOSUB 2000
1899 RETURN

1900 REM menu7
1920 DATA "Next obj.","Cycle","Delete","Exit"
1930 D%=4
1940 RESTORE 1900
1950 GOSUB 2000
1999 RETURN

2000 REM data
2010 CLS#1
2015 PRINT#1
2020 FOR I%=1 TO D%
2030  READ M$
2040  PRINT #1,M$
2050  PRINT #1
2060 NEXT I%
2099 RETURN

2500 REM update
2510 I%=0:J%=0
2520 IF INKEY(0)=0 THEN I%=I%+1
2530 IF INKEY(2)=0 THEN I%=I%-1
2540 IF INKEY(8)=0 THEN J%=J%-1
2550 IF INKEY(1)=0 THEN J%=J%+1
2560 IF INKEY(3)=0 THEN I%=I%*10:J%=J%*10
2570 X%=X%+J%
2580 Y%=Y%+I%
2599 RETURN
```

```
3000 REM draw cursor
3010 PRINT CHR$(23);CHR$(1);
3020 MOVE XG%,YG%+8
3030 DRAWR 0,-16
3040 MOVE XG%+8,YG%
3050 DRAWR -16,0
3060 PRINT CHR$(23);CHR$(0);
3070 LOCATE#1,1,22:PRINT#1,"X=";XG%;SPC(1);
3080 LOCATE#1,1,23:PRINT#1,"Y=";YG%;SPC(1);
3099 RETURN

3200 REM object
3210 GOSUB 1700
3220 IF INKEY(62)=0 THEN GOSUB 3300:GOSUB 1300:RETURN
3230 IF INKEY(61)=0 THEN GOSUB 4200:GOSUB 1300:RETURN
3240 IF INKEY(51)=0 THEN GOSUB 8600:GOSUB 1300:RETURN
3250 GOTO 3220

3300 REM create
3310 GOSUB 9000
3315 OM%=0
3316 GOSUB 3500
3320 OB%=OB%+1
3330 GOSUB 1800
3335 PRINT CHR$(23);CHR$(1);
3340 WHILE INKEY(58)<>0
3350  DLI%=S%:GOSUB 3700
3360  IF INKEY(62)=0 THEN GOSUB 3500:K%=62:GOSUB 9800
3370  IF INKEY(69)=0 THEN GOSUB 3800:GOSUB 3500:K%=69:
      GOSUB 9800
3380  DLI%=S%:GOSUB 3700
3390 WEND
3400 DLI%=S%:GOSUB 3700
3410 K%=58:GOSUB 9800
3420 PRINT CHR$(23);CHR$(0);
3499 RETURN

3500 REM lcycle
3510 DLI%=S%:GOSUB 3700
3515 LE%=0
3520 S%=S%+1:LE%=LE%+1
3530 IF S%>L% THEN S%=1
3535 IF O%(S%)<>OM% AND LE%<L% THEN GOTO 3520
3540 GOSUB 9500
3550 DLI%=S%:GOSUB 3700
3560 GOSUB 9800
3599 RETURN

3700 REM dline
3710 MOVE X%(S%(DLI%)),Y%(S%(DLI%))
3720 DRAW X%(E%(DLI%)),Y%(E%(DLI%))
3799 RETURN
```

```
3800 REM ostore
3810 O%(S%)=OB%
3820 MOVE X%(S%(S%)),Y%(S%(S%))
3830 DRAW X%(E%(S%)),Y%(E%(S%))
3840 GOSUB 9500
3850 GOSUB 9800
3899 RETURN

4000 REM command
4010 IF INKEY(27)=0 THEN GOSUB 5000:K%=27:GOSUB 9800
4020 IF INKEY(36)=0 THEN GOSUB 8000:K%=36:GOSUB 9800
4030 IF INKEY(61)=0 THEN GOSUB 9000:GOSUB 1300:K%=61:
     GOSUB 9800
4040 IF INKEY(38)=0 THEN GOSUB 7000:K%=38:GOSUB 9800
4050 IF INKEY(34)=0 THEN GOSUB 3200:K%=34:GOSUB 9800
4060 IF INKEY(53)=0 THEN GOSUB 6000:K%=53:GOSUB 9800
4070 IF INKEY(58)=0 THEN GOSUB 5100:K%=58:GOSUB 9800
4080 IF INKEY(59)=0 THEN GOSUB 7700:K%=59:GOSUB 9800
4090 IF INKEY(67)=0 THEN GOSUB 9900:K%=67:GOSUB 9800
4199 RETURN

4200 REM drag
4210 GOSUB 4400
4220 GOSUB 9000
4230 GOSUB 1500
4240 WHILE INKEY(58)<>0
4250  GOSUB 4600
4260  IF INKEY(62)=0 THEN GOSUB 9500:GOSUB 8200:K%=62:
      GOSUB 9800
4270  X%=0:Y%=0
4280  GOSUB 2500
4290  GOSUB 4800
4300  GOSUB 4600
4310 WEND
4320 X%=XG%:Y%=YG%
4330 K%=58:GOSUB 9800
4399 RETURN

4400 REM ocycle
4410 N%=0
4420 OM%=OM%+1
4430 IF OM%>OB% THEN OM%=0
4440 FOR I%=1 TO L%
4450   IF O%(I%)<>OM% THEN GOTO 4480
4460   N%=N%+1
4470   T%(N%)=I%
4480 NEXT I%
4490 M%=0
4495 IF N%=0 THEN GOTO 4594
4500 FOR R%=1 TO N%
4510  J%=S%(T%(R%)):K%=E%(T%(R%))
```

```
4520  F1%=0:F2%=0
4530  FOR I%=1 TO M%
4540   IF Q%(I%)=J% THEN F1%=1
4550   IF Q%(I%)=K% THEN F2%=1
4560  NEXT I%
4570  IF F1%=0 THEN M%=M%+1:Q%(M%)=J%
4580  IF F2%=0 THEN M%=M%+1:Q%(M%)=K%
4590 NEXT R%
4594 IF OM%=0 THEN GOSUB 4700
4595 IF M%=0 THEN GOTO 4400
4599 RETURN

4600 REM draw object
4605 PRINT CHR$(23);CHR$(1);
4610 IF N%=0 THEN GOTO 4699
4620 FOR I%=1 TO N%
4630  DLI%=T%(I%):GOSUB 3700
4640 NEXT I%
4650 PRINT CHR$(23);CHR$(0);
4699 RETURN

4700 REM add points to object 0
4710 FOR I%=1 TO P%
4720  J%=1
4730  WHILE S%(J%)<>I% AND E%(J%)<>I% AND J%<=L%
4740   J%=J%+1
4750  WEND
4760  IF J%>L% THEN M%=M%+1:Q%(M%)=I%
4770 NEXT I%
4799 RETURN

4800 REM drag
4810 IF M%=0 THEN GOTO 4899
4820 FOR I%=1 TO M%
4830  X%(Q%(I%))=X%(Q%(I%))+X%
4840  Y%(Q%(I%))=Y%(Q%(I%))+Y%
4850 NEXT I%
4899 RETURN

5000 REM point
5010 P%=P%+1
5020 X%(P%)=XG%:Y%(P%)=YG%
5030 PLOT XG%,YG%
5040 GOSUB 9500
5099 RETURN

5100 REM erase
5110 GOSUB 9000
5120 GOSUB 1900
5130 K%=58:GOSUB 9800
5134 GOSUB 4400:GOSUB 3500
5135 PRINT CHR$(23);CHR$(1);
```

```
5140 WHILE INKEY(58)<>0 AND L%>0
5150  DLI%=S%:GOSUB 3700
5155  IF INKEY(46)=0 THEN GOSUB 4400:GOSUB 3500:K%=46:
      GOSUB 9800
5160  IF INKEY(62)=0 THEN GOSUB 3500:K%=62:GOSUB 9800
5170  IF INKEY(61)=0 THEN GOSUB 5300:DLI%=S%:
      GOSUB 3700:K%=61:GOSUB 9800
5180  DLI%=S%:GOSUB 3700
5200 WEND
5210 GOSUB 1300
5220 K%=58:GOSUB 9800
5230 PRINT CHR$(23);CHR$(0);
5299 RETURN

5300 REM lerase
5310 I%=S%(S%):J%=E%(S%)
5320 IF I%<J% THEN F1%=I%:I%=J%:J%=F1%
5330 FOR K%=S% TO L%-1
5340  S%(K%)=S%(K%+1)
5350  E%(K%)=E%(K%+1)
5360  O%(K%)=O%(K%+1)
5370 NEXT K%
5380 L%=L%-1
5390 IF S%>L% THEN S%=1
5400 F1%=0:F2%=0
5410 FOR K%=1 TO L%
5420  IF I%=S%(K%) OR I%=E%(K%) THEN F1%=1
5430  IF J%=S%(K%) OR J%=E%(K%) THEN F2%=1
5440 NEXT K%
5450 IF F1%=0 THEN RI%=I%:GOSUB 5500
5460 IF F2%=0 THEN RI%=J%:GOSUB 5500
5499 RETURN

5500 REM remove
5510 FOR K%=RI% TO P%-1
5520  X%(K%)=X%(K%+1)
5530  Y%(K%)=Y%(K%+1)
5540 NEXT K%
5550 FOR K%=1 TO L%
5560  IF S%(K%)>RI% THEN S%(K%)=S%(K%)-1
5570  IF E%(K%)>RI% THEN E%(K%)=E%(K%)-1
5580 NEXT K%
5590 P%=P%-1
5599 RETURN

6000 REM file
6010 GOSUB 1600
6020 REM loop back
6030  IF INKEY(60)=0 THEN GOSUB 6200:GOTO 6070
6040  IF INKEY(36)=0 THEN GOSUB 6400:GOTO 6070
6050  IF INKEY(69)=0 THEN GOSUB 6600:GOTO 6070
6060 GOTO 6020
```

```
6070 GOSUB 9000
6080 GOSUB 1300
6099 RETURN

6200 REM save
6210 GOSUB 6900
6220 OPENOUT F$
6230 WRITE#9,P%,L%,OB%
6240 FOR I%=1 TO P%
6250  WRITE#9,X%(I%),Y%(I%)
6260 NEXT I%
6270 FOR I%=1 TO L%
6280  WRITE#9,S%(I%),E%(I%),O%(I%)
6290 NEXT I%
6300 CLOSEOUT
6399 RETURN

6400 REM load
6410 GOSUB 6900
6420 OPENIN F$
6430 INPUT#9,P%,L%,OB%
6440 FOR I%=1 TO P%
6450  INPUT#9,X%(I%),Y%(I%)
6460 NEXT I%
6470 FOR I%=1 TO L%
6480  INPUT#9,S%(I%),E%(I%),O%(I%)
6490 NEXT I%
6500 CLOSEIN
6599 RETURN

6600 REM append
6610 GOSUB 6900
6620 OPENIN F$
6630 INPUT#9,A%,B%,D%
6640 FOR I%=P%+1 TO P%+A%
6650  INPUT#9,X%(I%),Y%(I%)
6660 NEXT I%
6670 FOR I%=L%+1 TO L%+B%
6680  INPUT#9,S%(I%),E%(I%),O%(I%)
6690  S%(I%)=S%(I%)+P%
6700  E%(I%)=E%(I%)+P%
6710  IF O%(I%)<>0 THEN O%(I%)=O%(I%)+OB%
6720 NEXT I%
6730 P%=P%+A%
6740 L%=L%+B%
6750 OB%=OB%+D%
6760 CLOSEIN
6799 RETURN

6900 REM filename
6910 WHILE INKEY$<>""
```

```
6920 WEND
6930 INPUT#1,"file name ";F$
6999 RETURN

7000 REM move
7010 GOSUB 9000
7015 GOSUB 1300
7016 GOSUB 4400
7020 GOSUB 8400
7030 GOSUB 9700
7040 WHILE INKEY(58)<>0
7045  IF INKEY(46)=0 THEN GOSUB 4400:GOSUB 8400:K%=46:
      GOSUB 9800
7050  IF INKEY(62)=0 THEN GOSUB 8400:K%=62:GOSUB 9800
7060  X%=XG%:Y%=YG%
7070  GOSUB 3000
7090  GOSUB 2500
7100  XG%=X%:YG%=Y%
7110  GOSUB 7500
7120  GOSUB 3000
7130  X%(Q%(C%))=XG%
7140  Y%(Q%(C%))=YG%
7150 WEND
7160 GOSUB 1300
7170 K%=58:GOSUB 9800
7199 RETURN

7500 REM band
7505 PRINT CHR$(23);CHR$(1);
7510 FOR I%=1 TO NP%
7520  DLI%=C%(I%):GOSUB 3700
7530 NEXT I%
7540 X%(Q%(C%))=XG%:Y%(Q%(C%))=YG%
7550 FOR I%=1 TO NP%
7560  DLI%=C%(I%):GOSUB 3700
7570 NEXT I%
7580 PRINT CHR$(23);CHR$(0);
7599 RETURN

7700 REM origin
7710 ORIGIN -XG%,-YG%
7720 GOSUB 9000
7730 GOSUB 1300
7799 RETURN

8000 REM line
8010 GOSUB 1400
8015 GOSUB 4400
8020 GOSUB 8400
8030 WHILE INKEY(53)<>0
8040  IF INKEY(62)=0 THEN GOSUB 8400:K%=62:GOSUB 9800
```

```
8045  IF INKEY(46)=0 THEN GOSUB 4400:GOSUB 8400:K%=46:
      GOSUB 9800
8050 WEND
8060 L%=L%+1
8070 S%(L%)=Q%(C%)
8075 OT%=OM%
8080 GOSUB 8400
8090 WHILE INKEY(60)<>0
8100  IF INKEY(62)=0 THEN GOSUB 8400:K%=62:GOSUB 9800
8105  IF INKEY(46)=0 THEN GOSUB 4400:GOSUB 8400:K%=46:
      GOSUB 9800
8110 WEND
8120 E%(L%)=Q%(C%)
8125 IF OM%=OT% THEN O%(L%)=OM%
8130 GOSUB 1300
8140 GOSUB 3000
8150 DLI%=L%:GOSUB 3700
8160 GOSUB 3000
8170 GOSUB 9500
8199 RETURN

8200 REM change object
8210 GOSUB 4600
8220 GOSUB 4400
8230 GOSUB 4600
8299 RETURN

8400 REM cycle
8405 IF M%=0 THEN GOSUB 4400
8410 C%=C%+1
8420 IF C%>M% THEN C%=1
8430 GOSUB 3000
8440 XG%=X%(Q%(C%))
8450 YG%=Y%(Q%(C%))
8460 GOSUB 3000
8470 GOSUB 9500
8480 GOSUB 8500
8499 RETURN

8500 REM find con
8510 NP%=0
8520 FOR I%=1 TO L%
8530  IF S%(I%)=Q%(C%) OR E%(I%)=Q%(C%) THEN
      NP%=NP%+1:C%(NP%)=I%
8540 NEXT I%
8599 RETURN

8600 REM object transformations
8610 GOSUB 4400
8620 GOSUB 8700
8630 GOSUB 9000
8640 GOSUB 9400
```

```
8650 GOSUB 4600
8660 IF INKEY(62)=0 THEN GOSUB 8200:GOSUB 8700:K%=62:
     GOSUB 9800
8670 IF INKEY(36)=0 THEN S=1.1:GOSUB 8800:K%=36:
     GOSUB 9800
8680 IF INKEY(60)=0 THEN S=0.9:GOSUB 8800:K%=60:
     GOSUB 9800
8690 IF INKEY(50)=0 THEN A=PI/8:GOSUB 8900:K%=50:
     GOSUB 9800
8694 IF INKEY(58)=0 THEN GOSUB 9000:GOSUB 1300:K%=58:
     GOSUB 9800:RETURN
8695 GOSUB 4600
8696 GOTO 8650

8700 REM find centroid
8710 XC%=0:YC%=0
8720 FOR I%=1 TO M%
8730  XC%=X%(Q%(I%))+XC%
8740  YC%=Y%(Q%(I%))+YC%
8750 NEXT I%
8760 XC%=XC%/M%:YC%=YC%/M%
8799 RETURN

8800 REM scale
8810 FOR I%=1 TO M%
8820  X%(Q%(I%))=(X%(Q%(I%))-XC%)*S+XC%
8830  Y%(Q%(I%))=(Y%(Q%(I%))-YC%)*S+YC%
8840 NEXT I%
8899 RETURN

8900 REM rotate
8910 FOR I%=1 TO M%
8920  XN=(X%(Q%(I%))-XC%)*COS(A)-(Y%(Q%(I%))-YC%)
      *SIN(A)+XC%
8930  YN=(X%(Q%(I%))-XC%)*SIN(A)+(Y%(Q%(I%))-YC%)
      *COS(A)+YC%
8940  X%(Q%(I%))=XN
8950  Y%(Q%(I%))=YN
8960 NEXT I%
8999 RETURN

9000 REM draw
9010 CLG
9030 GOSUB 9600
9040 FOR I%=1 TO L%
9050  DLI%=I%:GOSUB 3700
9060 NEXT I%
9070 GOSUB 3000
9099 RETURN

9400 REM menu8
9410 DATA "Cycle","Larger","Smaller"
```

```
9420 DATA "Rotate","Exit"
9430 RESTORE 9400
9440 D%=5
9450 GOSUB 2000
9460 RETURN

9500 REM bell
9510 PRINT CHR$(7);
9599 RETURN

9600 REM plot points
9610 FOR I%=1 TO P%
9620  PLOT X%(I%),Y%(I%)
9630 NEXT I%
9699 RETURN

9700 REM menu9
9710 DATA "Next obj.","Cycle","Exit"
9720 D%=3
9730 RESTORE 9700
9740 GOSUB 2000
9799 RETURN

9800 REM clear key press
9810 WHILE INKEY(K%)=0
9820 WEND
9899 RETURN

9900 REM QUIT
9910 WHILE INKEY$<>""
9920 WEND
9930 LOCATE#1,1,20
9940 INPUT#1,"ARE YOU    SURE ?",A$
9950 IF A$="Y" THEN END
9960 GOSUB 9000
9970 GOSUB 1300
9999 RETURN
```

After the fairly full descriptions of the way that point and line data is input, stored and edited, the graphics editor should be fairly easy to understand. A text window is set up on the left hand side of the screen so that menus listing possible commands can be displayed to the user. You can draw under the menu area but what you draw will be obliterated (though not lost) when a menu is printed. You can draw anywhere on or off the screen as the area that is displayed is only a window onto a larger drawing. Initially, a small cross standing for the graphics cursor is displayed and can be moved around the screen using the cursor keys. When a key that corresponds to the first letter of any command displayed in the current menu is pressed then the program enters a different mode depending on the action required. The commands implemented are

P – enter a new Point
L – enter a new Line
D – Draw the current data
M – Move a point
O – create, drag or transform an Object
F – Save, load or append a graphics File
E – Erase a line
W – Window
Q – Quit

Pressing P enters the co-ordinates of the graphics cursor in X% and Y% as described in the previous section. The 'Line' command produces a new menu on the screen and three new commands

N – Next object
C – Cycle to next point in current object
F – First point of line
S – Second point of line

The 'Draw' command draws all of the lines and points that have been entered, so producing a 'clean' display. The 'Move' command produces a third menu containing the commands

N – Next Object
C – Cycle to next point in current object
E – Exit back to main menu

While this menu is displayed you can move any existing point by pressing C and N until the graphics cursor is positioned over it and then using the arrow keys to shift its current position.

The 'Object' command produces a new menu which allows the user either to create a new object, or transform or drag an existing object.

C – Create new object
D – Drag object
T – Transform object

Each time an object is created it is assigned the next available object number. New lines that are drawn between two points that belong to the same object are appended to that object, otherwise new lines are assigned to object zero. The lines that make up a new object are determined by cycling through the lines that are currently assigned to object zero. Dragging an object works in a manner analogous to rubber banding. The object to be dragged is determined by cyclic selection and then moved into place using the arrow keys. Pressing 'T' produces a further new menu offering the following options

C – Cycle around the objects
L – make the selected object Larger
S – make the selected object Smaller
R – Rotate the selected object
E – Exit

The 'C' command permits the user to select which object is to be transformed by cycling through the objects. Pressing 'L' makes the currently selected object get bigger and pressing 'S' makes it get smaller. In other words, 'L' and 'S' provide an implementation of a full scaling transformation. Pressing 'R' once makes the object rotate through 22½ degrees (that is 1/16th of a revolution). All of these transformations are carried out interactively in the sense that the object is redrawn after the transformation so that the user can see what is happening. For example, to produce a rotation of 90 degrees the user would press 'R' four times and after each keypress the object would be displayed. Notice that because of inaccuracies and rounding errors the shape of an object may change under repeated scaling or rotation.

The 'File' command will save all of the arrays that define the current display. These can then be loaded or appended to existing graphics data. Saving and appending can be used to generate quite complex displays very quickly. For example, suppose you define a square, then by saving it and appending it you can build up a display with any number of squares, which can then be transformed, rotated etc. As long as you save single objects defined as object number 1, appending objects results in them receiving new object numbers just as if they had been produced using the 'Create' option.

The 'Erase' command works by allowing the user to select a line by the usual method of cyclic object selection and then delete it from the line file.

The 'Window' option is the only one that hasn't been described earlier. If the window option is selected the current position of the graphics cursor is made to correspond to the bottom left hand corner of the screen and this results in a different portion of the drawing becoming visible. Using the window option you can have a drawing as large as you like, but be warned, object cycling and all operations involve all of the objects and points even if they cannot be seen because of the current position of the origin! To restore the origin to its usual place in the bottom left hand corner of the screen move the graphics cursor to 0,0 (using the co-ordinate display in the menu if necessary) and select the window option. Finally the 'Quit' option allows you to stop the program!

A description of each subroutine is given below

Subroutine use – AMPLOT

Subroutine	function
1000	Sets up variable and arrays
1200	Defines a text window
1300	Main menu
1400	Line menu
1500	Drag menu
1600	File menu
1700	Object menu
1800	Create object menu
1900	Erase line menu
2000	Prints a menu
2500	Updates cursor position
3000	Draws cursor at XG%,YG%
3200	Selects, creates or drags object
3300	Creates object by cycling around lines assigned to object zero and storing object number OM%in O%
3500	Cycles around lines in current object
3700	Draws line DLI%
3800	Stores object number in O%
4000	Selects command from main menu
4200	Drags object
4400	Cycles around objects and builds line list in T% and point list in Q%
4600	Draws object defined by the line list in T%
4700	Add isolated points to object 0
4800	Updates points in point list Q%
5000	Adds point to point file
5100	Erases a line
5300	Removes line from line file and any isolated points from the point file
5500	Removes point from point file and corrects line file accordingly
6000	Selects load, save or append
6200	Saves a graphics file
6400	Loads a graphics file

6600	Appends a graphics file
6900	Gets file name in F$
7000	Moves a point
7500	Performs rubber banding
7700	Shifts origin to current position of graphics cursor
8000	Adds a line to the line file
8200	Changes object
8400	Cycles around points in current object
8500	Finds all the lines that reference point C%
8600	Selects object transformation
8700	Finds centroid of current object
8800	Scales object about centroid, scale factor=S
8900	Rotates object about centroid
9000	Draws all lines in line file
9400	Object transformations menu
9500	Rings bell
9600	Draws all the points in the point file
9700	Move point menu
9800	Waits until key K% is released
9900	Quits program

From its description AMPLOT may sound complicated to use, but the inclusion of menus makes it quite simple in practice. It gives the user a great deal of freedom and flexibility and by building up a library of shapes can create complex diagrams and technical drawings quickly and efficiently.

7 Looking at three dimensions

Three-dimensional computer graphics are exciting because they are a way of seeing objects which do not exist, and may never exist! So many films are now using backgrounds and even characters that are entirely the figment of a computer's imagination that realistic three-dimensional computer graphics are becoming commonplace. This is not to say that creating such displays is easy. Some of the most powerful computers ever built are needed to produce a few seconds of film and even this can take them days! With this in mind you might think that trying to produce three-dimensional displays using BASIC would be far too difficult. This is true if your aim is to produce the sort of realistic graphics to be seen in feature films. However, it is possible to write a program in BASIC that will allow you to view a 'wire frame' representation of an object from any position. For many purposes this limited form of representation is quite sufficient and the rest of this chapter is concerned with developing a wire frame viewer.

Micro 3D

Recent micro-computer games have tended to give the impression that not only is it easy to produce simple three-dimensional views, it is even easy to make three-dimensional objects move. However, the sort of three-dimensional graphics that occurs in games has more in common with the way an artist or draughtsman uses lines on a flat sheet of paper to produce the appearance of depth. True three-dimensional computer graphics uses a complete description of a shape to produce a two-dimensional representation of what would be seen from any particular 'viewpoint'. For example, you might record all of the information concerning the shape of a house and then choose to view it from above, giving a sort of floor plan, or from the front, say. The important point is that in true three-dimensional graphics the viewpoint can be changed with complete freedom thus allowing the user to examine the object and gain a good impression of what it is like. On the other hand, the 'artistic' method of three-dimensional graphics would use ideas similar to those introduced in Chapters Five and Six concerning two-dimensional graphics to display a floor plan or a front elevation of the house, or whatever, in much the same way that you would draw the same view using pencil and paper. Although such drawings can be made 'free-hand' using nothing by intuition there are principles that govern three-dimensional sketching. If you are interested in learning more about art in three dimensions then see 'Graphical Communication', (1985), by A. Yarwood, published by Hodder & Stoughton.

Wire frame and solid face representation

The point and the line play an equally important role in three-dimensional graphics as they do in two-dimensional. In two dimensions two points define the location of a straight line and a number of connected lines define a shape. The same is true in three-dimensional graphics, the only difference is that each point requires three co-ordinates, usually denoted x,y,z. However the most important feature of three-dimensional graphics is that the two-dimensional shapes defined by a number of connected lines can be used to enclose a volume and hence form a solid shape. For example, a cube can be defined by putting together six square faces, each defined by four lines.

There are two slightly different types of three-dimensional representation, depending on whether or not the faces of a solid object are considered transparent or opaque. If they are transparent, then all the faces of an object are visible at the same time and it appears as a 'wire frame' (see fig 7.1a). If the faces are opaque them some of the faces of the object will be obscured by others that are closer to the viewing position (see fig 7.1b). Producing three-dimensional images with opaque faces is a very difficult problem involving so called 'hidden line removal' algorithms. Although detecting and removing lines that are hidden from view by other faces seems like a trivial problem it is in fact extremely time-consuming and well outside the practical range of things that the current micros can do without long time delays. For this reason the rest of this chapter concentrates on producing a wire frame representation of an object but it is worth saying that this is usually the first stage on the way to producing a solid face representation.

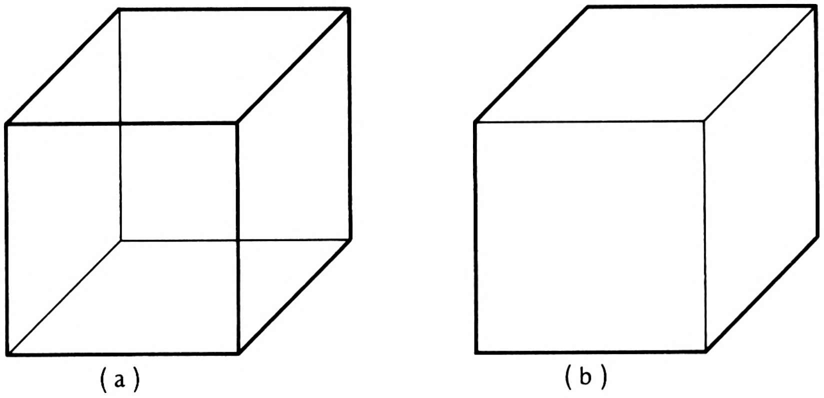


Fig 7.1 a) transparent cube and b) solid cube after hidden line removal

Point, line and face files

The problem of finding a data representation suitable for three-dimensional graphics is more complex than you might expect. Obviously the method of point and line files used in two-dimensional graphics can easily be extended to three dimensions by adding an extra co-ordinate to each point in the point file. However, although the point and line file are sufficient to define an object, they do not always present the information needed in the most accessible form. For example, given the co-ordinates of the eight corners of a cube stored in the point file and the 12 edges in the line file, it is a difficult job to work out which lines in the line file go together to form each face of the cube. This difficulty could be alleviated by adding a 'face file' to the data structures. Each entry in the face file would simply be a list of the lines in the line file that make up each face. Once the face file has been added, it is easy to see that it might be worth recording other information about each face at the same time. For example its colour or texture could be used to create a shaded solid representation (given suitable hardware!). Fortunately wire frame representations of objects can be constructed without any additional complications to the point and line files and the problem of identifying faces will be ignored for the rest of this chapter.

The 3D data explosion

Now that the method of storing the data that defines a three-dimensional object has been decided, it is worth giving an example that illustrates the 'data explosion' problem of three-dimensional graphics. The points that have to be stored in the point file to represent the corners of a 'unit cube' can be seen in

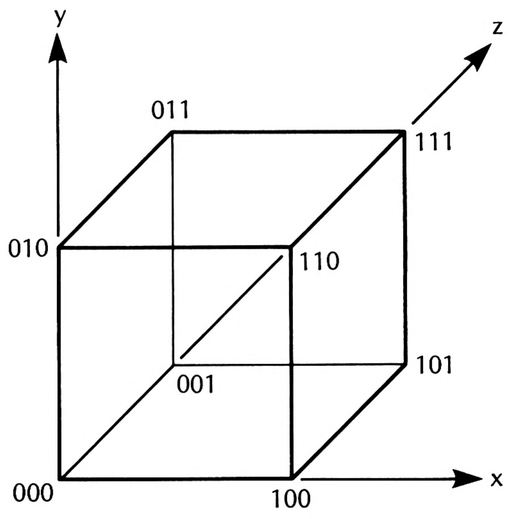


Fig 7.2 Unit cube with x,y and z co-ordinates

fig 7.2. The three numbers written next to each corner are the x co-ordinate, y co-ordinate and the z co-ordinate respectively. Thus, the point file consists of three arrays X, Y and Z and the Ith point is given by X(I), Y(I) and Z(I). It is obvious that even a simple cube needs 24 (8 corners * 3 co-ordinates) array elements to represent them. There are also twelve lines that have to be stored in the line file to complete the description of the cube. Each line is defined by a start point and an end point and so the twelve lines take 24 array elements to store, making a grand total of 48 array elements. This should be compared to the 16 elements needed in two dimensions to store the description of a square. It doesn't take much imagination to see that getting the data for an object with a complex shape into the system is going to be a major problem.

Three-dimensional transformations

The method of rotating, translating and scaling in two dimensions is easy to extend to three dimensions. The only real change is that instead of transforming a pair of co-ordinates we have to transform three and hence use three equations. For example a three-dimensional translation is produced by the following three equations

$$x' = x + a$$
$$y' = y + b$$
$$z' = z + c$$

where the values of a, b and c control the amount and direction that each point is moved. In the same way, a three-dimensional scaling can be produced by multiplying each co-ordinate by a scaling factor. Three-dimensional rotations are rather more tricky than their two-dimensional counterparts, in that it is possible to rotate a three-dimensional object about any given line. In other words, in three dimensions objects rotate about 'axes of rotation' rather than points of rotation. It is possible to write down a transformation for rotation about a general axis but it is very complicated. Fortunately, a three-dimensional rotation about any given line can be produced by a combination of rotations about each of the three co-ordinate axes. This is analogous to the way a rotation about any given point can be produced using a rotation about the origin in two-dimensional graphics, see Chapter Six. Rotations about the co-ordinate axes are comparatively simple. For example, a rotation around the Z axis through an angle T is produced by

$$x' = x\cos(t) - y\sin(t)$$
$$y' = x\sin(t) + y\cos(t)$$
$$z' = z$$

Rotations about the other axes can be produced by similar sets of equations. If you would like to know more about how to derive three-dimensional transformations, you will have to learn something about matrix algebra. There are lots of books on this subject but one relevant source is 'Principles of Interactive Computer Graphics, 2nd Edition', (1981), by W. M. Newman and R. F. Sproull, published by McGraw-Hill.

Projections

Using three-dimensional transformations, it is possible to alter the view of any object by rotation, translation or scaling but there still remains a fundamental problem – how can a three-dimensional object be drawn on a two-dimensional screen. This problem is essentially that each point on the object has three co-ordinates but a point on the display screen has only two co-ordinates. Clearly what is needed is a way of reducing the three co-ordinates to two in such a way that the points plotted on the display screen look like the object when viewed from the given position. In other words, we have to find a transformation that will convert each point in the object x,y,z to a point a,b in two dimensions, such that the set of two-dimensional points gives an impression of the original object. A transformation of this type is known as a 'projection' because it is related to the idea of projecting a shadow of an object onto a screen.

There are a number of different types of projection, but the only one that produces a realistic view of an object is the 'perspective projection'. In real life the apparent size of something depends on how far away it is from the observer. For example, if you were to view a wire cube from a few feet the top front horizontal edge would appear slightly larger than the top rear horizontal edge – see fig 7.3. As you move further away from the cube then the difference between the two edges becomes less, and as you move closer it becomes exaggerated. This is, of course, the phenomenon of 'perspective' and it is something that we all take for granted both in our everyday viewing of the world and in traditional pictures and drawings that attempt to produce a sense of realism.

The equations of the perspective projection of the point given by x,y,z into two dimensions is surprisingly simple

$$a = \frac{z\ xc - x\ zc}{z - zc}$$

$$b = \frac{z\ yc - y\ zc}{z - zc}$$

where the point given by xc,yc,zc is called the 'centre of projection'. Roughly speaking, the centre of projection is the position that an observer would occupy to see the same results as those produced by the perspective projection. The perspective projection produces a two-dimensional representation rather like a photograph of a wire frame model of the object, and the centre of projection is the position where an observer would have to be to

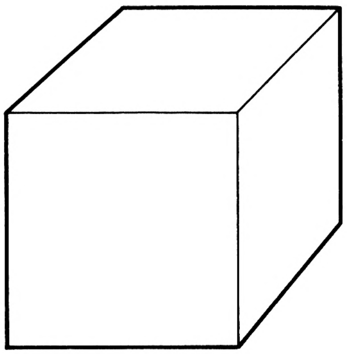


Fig 7.3 The effects of perspective

see exactly what the photograph shows. Obviously, the position of the centre of projection alters the amount of each side of the cube which can be seen in the projection, but how does its distance from the cube affect the projection? If the centre of projection is moved further away the perspective effect is lessened and if it is moved closer the perspective is exaggerated – and this is exactly what the observer would see if located at the centre of projection. For any viewed object there is a position of the centre of projection that will produce a realistic display: too close to the object and the perspective will appear exaggerated, too far away and the perspective will be too slight.

The data entry problem

The data explosion problem in three-dimensional graphics makes high demands on the amounts of computer memory needed to store even a simple object, but there is also the problem of entering such large amounts of data. In the two-dimensional case it was possible to avoid entering long lists of co-ordinate, line and object data by using interactive data entry and modification. In principle this interactive approach could be extended to include three-dimensional graphics but it presents substantial difficulties at two levels.

First, there is the obvious problem of making the program work fast enough to give the impression of real time data entry and editing. Even if these problems of speed of calculation could be overcome, there is the second and more fundamental problem of the ambiguity of the third co-ordinate. A projection reduces to a point in three dimensions, at x,y,z say, to a point in two dimensions at a,b. However, selecting this point on the screen doesn't automatically specify the point x,y,z in three dimensions. There are a great many points in three dimensions that are projected onto the point a,b and to let the computer know which one you are referring to you have to supply some additional information. In other words, there is no simple way of inverting a projection so that you can go back from screen co-ordinates to three-dimensional co-ordinates. Of course, there are ways of providing the additional information to determine which of the many three-dimensional points you are referring to but they are all unnatural and clumsy. There seems to be no alternative but to accept that while the TV screen is an adequate three-dimensional output device it is poor when it comes to the associated input task.

An alternative method of building up a three-dimensional object is to use the draughtsman's traditional technique of drawing a number of different 'elevations' of an object. After a little practice humans are remarkably good at putting together the different pieces of information contained in, say, a top, front and side elevation of an object to arrive at a three-dimensional understanding of the object. A computer can also be programmed to take a number of elevations and fuse them into a three-dimensional object, but it is more difficult than you might expect. A system based on this idea would allow the user to modify points displayed in three separate elevations of the object while simultaneously displaying a three-dimensional representation of the object in the middle of the screen. Once again the computational requirements are well beyond anything that a micro can handle in a reasonable time. It is unfortunately true that in most cases the only way to obtain a three-dimensional representation of something inside a computer is to measure and enter the co-ordinates of each point, along with all the line and face information that is necessary – and this is often a major task.

Symmetry and solids of revolution

Although the prospect of entering large amounts of data to define a three-dimensional object is generally rather bleak there are certain special classes of object that can be generated from very small quantities of initial data. In particular, it is possible to take advantage of any symmetries an object might possess to reduce the total amount of data input required. For example if an object has a central plane of mirror symmetry – if it can be thought of as two halves, each a mirror image of the other – then the amount of data entry can be halved. After entering the points and lines that define one side of the object, an appropriate reflection transformation can be used to generate the other half.

An extreme example of the use of symmetry to reduce the data entry problem can be found in a class of objects that possess an axis of rotational symmetry – the so called 'bodies of revolution'. The idea of a body of revolution can be most easily understood by thinking of the solid objects that could be produced by cutting out a flat shape, in cardboard say, and rotating it around a central axis. For example the outline in fig 7.4, if rotated about its longest side would generate a wine glass shape. Obviously, a solid of revolution can be defined by entering only the co-ordinates of the points that make up its outline and this is considerably fewer than the total number of points it contains.

The wire frame viewer – AMVIEW

AMVIEW, the wire frame viewer program given below is, in

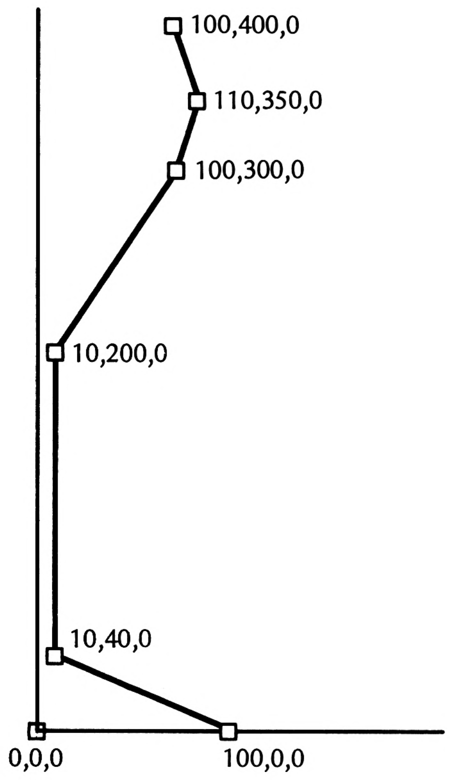


Fig 7.4 The profile of a wine glass

many ways, the first step on the way to the ultimate goal of three-dimensional graphics. That is, a program that will display a realistic representation of an object viewed from any position. AMVIEW will, given a definition of an object in terms of co-ordinates, allow you to view the object from any position, but only as a wire frame object. In terms of practical applications, a wire frame viewer is at the heart of such things as flight simulators and many computer-aided design packages. For example, in the case of the flight simulator the object being viewed is composed of all the buildings, roads, trees etc that make up the landscape, and the particular view that is produced on the computer screen corresponds to what would be seen from a aircraft flying over the landscape.

The wire frame viewer given in this chapter is not quite up to the standard of a flight simulator – but only because it is written in BASIC and hence not quite fast enough! In fact the speed of the program is much faster than you might expect for such a complex calculation – around a second per twenty display points – and it is almost good enough to give the impression of 'flying over the object being viewed. The response of the program could certainly be improved by optimising the procedures that do the calculation, but for real speed you cannot beat assembler.

The ideal view

Before presenting the wire frame viewer it is worth considering what an ideal program should allow the user to do. The difficulty of defining a three-dimensional object is simply the sheer number of co-ordinates that have to be entered. This, however, is not a problem that has much to do with the essence of the viewer and so the co-ordinates that define the object can be stored as DATA statements. Given a description of the object in terms of the co-ordinates of its points, the only item of data left to define is the position of the 'observer'. This is not as simple as you might expect. To start with it is not enough to give just the co-ordinates of the observer's position because, while this would tell you where the observer was, it wouldn't tell you where the observer was looking! It is clear that, even if the co-ordinates of the observer's position were sufficient, this wouldn't be an acceptable way for the user to interact with the program. What is needed is some way in which the user can control the movement of the observer in a 'natural' way, without having to worry about numerical co-ordinates. In general, when viewing a real object humans tend to think in terms of moving closer or further away, of moving to the right or the left or up and down and these are the three types of movement that the program should try to make available to the user.

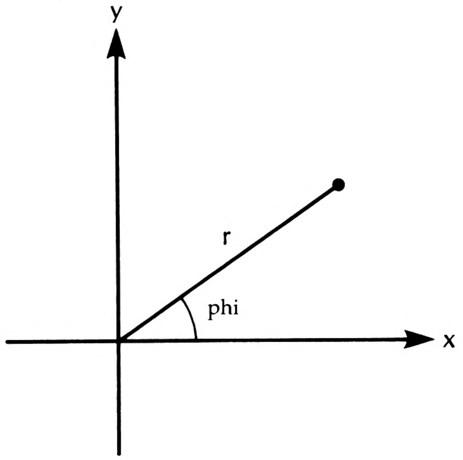


Fig. 7.5 R and phi as distance and angle of elevation

Moving the observer

It soon becomes obvious that the usual three x,y,z (or Cartesian) co-ordinates are not appropriate for the purpose of varying the observer's position. However, if the observer's position is specified using 'polar co-ordinates' then there is an immediate correspondence between the co-ordinates of a point and the distance from the object, moving left and right and up and down. In polar co-ordinates the position of a point is specified by giving three quantities: R, a distance; and two angles, TH and PH. (More usual notation than TH and PH are the Greek symbols theta and phi.) R is simply the distance of the point from the origin and PH is its angle elevation, see fig 7.5. The angle TH can be thought of as the number of degrees that you have to rotate the x axis through to reach the point, see fig 7.6. Keeping TH and PH fixed and varying R takes the point closer and further away from the origin without altering its orientation. Changing TH while keeping PH and R fixed makes the point 'move around' the origin without altering its distance or its height. Similarly, changing PH while keeping TH and R fixed alters only the height of the object but not its distance from the origin.

If the observer's position is specified using R, TH and PH then it is not difficult to see that each co-ordinate can be changed by pressing a key on the keyboard, thus giving the user a natural and interactive control of the observer's position. Although polar co-ordinates solve the problem of specifying the observer's position there is still the small matter of where the observer is looking. If we imagine that the observer is in fact a camera then the question can be more simply put as which way is the camera pointing? The most obvious solution is to make the camera look at

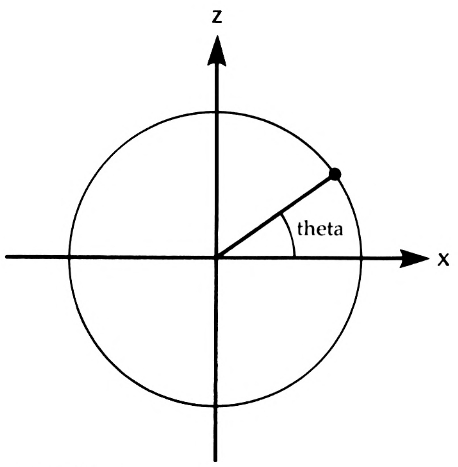


Fig 7.6 Theta as rotation around object

the origin, because this is the point that the polar co-ordinate system is centred on. The only trouble is that the object of interest may not be positioned at the origin! The answer is to shift the origin of the polar co-ordinate system to a position that corresponds to a point within the object that we want the camera to look toward at all times – this point is known as the 'centre of attention'. Thus the observer is best thought of as a camera that is always pointing at the centre of attention and its position, distance, rotation and elevation are all measured from this one important point.

The observer's view

Now that we have a way of specifying where the observer is and which way he is looking the rest is easy – in theory at least! The only remaining problem is that we have a description of the object in terms of co-ordinates that are relative to a fixed set of axes – the object's co-ordinate system – but these are not the co-ordinates that describe the object as viewed from the observer's position. Clearly, the observer has a different set of axes, that the co-ordinates of the object are measured from – the view co-ordinate system. The co-ordinates of the object in the view co-ordinate system represent the particular point of view that the observer has of the object and clearly they change as the position of the observer and the centre of attention change. To allow the program to show different views of the object, it is crucial to find a way of converting the unchanging object co-ordinates into the variable view co-ordinates – in fact what we need is a transformation.

Although it may sound difficult to work out, such a transformation involves no new principles. For example, as the observer moves around an object it appears to rotate and so its co-ordinates in the observer's co-ordinate system can be worked out by applying an appropriate transformation to produce a rotation about the object's z axis. In other words, it doesn't really matter whether you consider the observer to be rotating around a fixed object or the object to be rotating around a fixed observer! Even though there are no new principles involved, working out the transformation that will convert the object's co-ordinates into the co-ordinates that are appropriate for the observer's current position at R,TH,PH is a very tedious exercise in algebra. Rather than going into the details of the equations, the transformation is presented as a number of subroutines within the wire frame viewer.

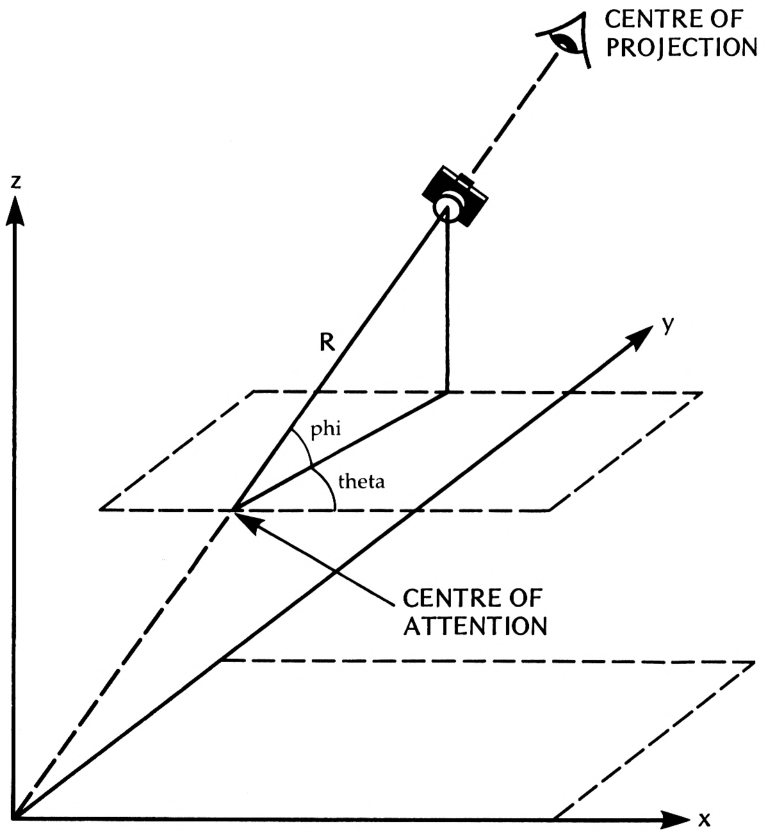


Fig 7.7 Co-ordinate system and the observer as a camera

A summary

At this point we have all of the ideas that are necessary to implement the wire frame viewer. The object or objects will be represented by point and line files as described earlier. The observer's position will be fixed by the location of the centre of attention and the polar co-ordinares R, TH and PH. To obtain the objects as seen from the observer's position all we have to do is apply the appropriate view transformation to all of the points in the point file. Finally to obtain a two-dimensional representation of the objects we have to apply a perspective transformation and then plot the resulting two-dimensional points and lines.

Positioning the centre of attention is simply a matter of deciding what point in the object you should look at. The appropriate position of the centre of projection is a little more difficult to determine. So far we have imagined the observer to be positioned at R,TH,PH and looking toward the centre of attention but in the section introducing the perspective projection it was stated that the observer could be thought of as positioned at the centre of projection. To understand exactly where the observer is it is better to think once again of the observer as a camera pointing at the centre of attention. The two-dimensional co-ordinates produced by the perspective transformation can be thought of as defining points on the film in the camera and the resulting image as a photograph of the scene. In these terms the centre of projection is the position that an observer would have to occupy to have the same view as that reproduced in the photograph. It is important to realise that the centre of projection is measured relative to the film in the camera with the x,y axis corresponding to the co-ordinates produced by the perspective projection. So it lies in the plane of the film and the z axis points toward the centre of attention, see fig 7.7. It is the convention that positive z co-ordinates correspond to positions on the object's side of the film.

AMVIEW

A complete listing of the wire frame viewer program can be seen below. It should come as something of a pleasant surprise to discover that such a sophisticated three-dimensional viewer turns out to be a fairly short BASIC program.

```
10    REM wire frame viewer
20    MODE 2
30    GOSUB 9000
40    GOSUB 1800
50    WHILE QUIT%=0
70     WHILE MENU%=0
80      IF M%=1 THEN GOSUB 1000 ELSE GOSUB 500
90      GOSUB 5000
100     GOSUB 6000
110     GOSUB 7000
120     GOSUB 3000
130     GOSUB 4000
140     GOSUB 2000
150    WEND
160    GOSUB 9000
170   WEND
199   END

500   REM data set for VDU
510   RESTORE 500
520   DATA 0,0,0
530   DATA 200,0,0
```

```
540  DATA 0,10,0
550  DATA 200,10,0
560  DATA 0,20,100
570  DATA 200,20,100
580  DATA 0,120,100
590  DATA 200,120,100
600  DATA 20,40,100
610  DATA 180,40,100
620  DATA 20,100,100
630  DATA 180,100,100
640  DATA 0,0,200
650  DATA 200,0,200
660  DATA 0,120,200
670  DATA 200,120,200
680  DATA 0,1
690  DATA 2,3
700  DATA 1,3
710  DATA 2,4
720  DATA 3,5
730  DATA 4,5
740  DATA 5,7
750  DATA 6,7
760  DATA 8,9
770  DATA 10,11
780  DATA 8,10
790  DATA 9,11
800  DATA 12,13
810  DATA 14,15
820  DATA 13,15
830  DATA 12,14
840  DATA 0,12
850  DATA 1,13
860  DATA 6,14
870  DATA 7,15
880  DATA 4,6
890  DATA 0,2
900  P=16
910  L=22
920  FOR I=0 TO P-1
930   READ X(I),Y(I),Z(I)
940  NEXT I
950  X(P)=0:Y(P)=1:Z(P)=0
960  P=P+1
970  FOR I=0 TO L-1
980   READ S(I),E(I)
990  NEXT I
995  CX=100:CY=60:CZ=100
999  RETURN

1000 REM data set for wine glass
1100 RESTORE 1000
1110 DATA 0,0,0
```

```
1120 DATA 100,0,0
1130 DATA 10,40,0
1140 DATA 10,200,0
1150 DATA 100,300,0
1160 DATA 110,350,0
1170 DATA 100,400,0
1180 DATA 0,1
1190 DATA 1,2
1200 DATA 2,3
1210 DATA 3,4
1220 DATA 4,5
1230 DATA 5,6
1240 P=7
1250 L=6
1260 FOR I=0 TO P-1
1270  READ X(I),Y(I),Z(I)
1280 NEXT I
1290 FOR I=0 TO L-1
1300  READ S(I),E(I)
1310 NEXT I
1320 N=10
1330 GOSUB 8000
1340 X(P)=0:Y(P)=1:Z(P)=0:P=P+1
1350 CX=0:CY=200:CZ=0
1799 RETURN

1800 REM init
1810 TH=PI/4
1820 PH=PI/180*35
1830 R=1000
1880 XC=320:YC=200
1890 ZC=-1000
1900 DIM A(200),B(200)
1910 DIM X(200),Y(200),Z(200)
1920 DIM S(400),E(400)
1930 QUIT%=0
1940 MENU%=0
1999 RETURN

2000 REM command
2010 WHILE A$<>"D" AND A$<>"M"
2020  LOCATE 1,24
2030  PRINT TAB(2);"DISTANCE=";R;
2040  PRINT TAB(20);"THETA=";INT(180*TH/PI);
2050  PRINT TAB(35);"PHI=";INT(180*PH/PI)
2060  A$=INKEY$:A$=UPPER$(A$)
2070  IF A$="" THEN GOTO 2060
2080  IF A$="C" THEN R=R-10
2090  IF A$="F" THEN R=R+10
2100  IF ASC(A$)=242 THEN TH=TH-PI/180
2110  IF ASC(A$)=243 THEN TH=TH+PI/180
```

```
2120  IF ASC(A$)=241 THEN PH=PH-PI/180
2130  IF ASC(A$)=240 THEN PH=PH+PI/180
2140  IF A$="M" THEN MENU%=1
2150 WEND
2160 A$=""
2999 RETURN

3000 REM pers
3010 FOR I=0 TO P-1
3020  A(I)=-X(I)*ZC+Z(I)*XC
3030  B(I)=-Y(I)*ZC+Z(I)*YC
3040  W=Z(I)-ZC
3050  A(I)=A(I)/W
3060  B(I)=B(I)/W
3070 NEXT I
3999 RETURN

4000 REM draw
4010 CLS
4020 FOR I=0 TO L-1
4030  MOVE A(S(I)),B(S(I))
4040  DRAW A(E(I)),B(E(I))
4050 NEXT I
4999 RETURN

5000 REM consts
5010 CT=COS(TH)
5020 ST=SIN(TH)
5030 CP=COS(PH)
5040 SP=SIN(PH)
5050 TX=-(CX+R*CP*CT)
5060 TY=-(CY+R*SP)
5070 TZ=-(CZ+R*CP*ST)
5080 CX=-CP*ST/SQR(SP*SP+CP*CP*ST*ST)
5090 SX=-SP/SQR(SP*SP+CP*CP*ST*ST)
5100 CY=SQR(SP*SP+CP*CP*ST*ST)
5110 SY=-CP*CT
5999 RETURN

6000 REM do trans
6010 FOR I=0 TO P-1
6020  IF I=P-1 THEN GOTO 6060
6030  X(I)=X(I)+TX
6040  Y(I)=Y(I)+TY
6050  Z(I)=Z(I)+TZ
6060  X=X(I)
6070  Y=Y(I)*CX-Z(I)*SX
6080  Z=Y(I)*SX+Z(I)*CX
6090  X(I)=X:Y(I)=Y:Z(I)=Z
6100  X=X(I)*CY-Z(I)*SY
6110  Y=Y(I)
6120  Z=X(I)*SY+Z(I)*CY
```

```
6130  X(I)=X:Y(I)=Y:Z(I)=Z
6140 NEXT I
6999 RETURN

7000 REM do upright
7010 V=SQR(X(P-1)*X(P-1)+Y(P-1)*Y(P-1))
7020 C=Y(P-1)/V
7030 S=X(P-1)/V
7040 FOR I=0 TO P-2
7050  X=X(I)*C-Y(I)*S
7060  Y=X(I)*S+Y(I)*C
7070  X(I)=X+320:Y(I)=Y+200
7080 NEXT I
7999 RETURN

8000 REM rotate (N)
8010 PN=P
8020 A=2*PI/N
8030 FOR J%=1 TO N-1
8040  CA=COS(A*J%)
8050  SA=SIN(A*J%)
8060  FOR I%=0 TO PN-1
8070   X(P)=X(I%)*CA
8080   Y(P)=Y(I%)
8090   Z(P)=X(I%)*SA
8100   P=P+1
8110   S(L)=P-1:E(L)=P
8120   L=L+1
8130  NEXT I%
8140  L=L-1
8150 NEXT J%
8160 FOR I%=0 TO PN-1
8170  FOR J%=1 TO N-1
8180   S(L)=I%+(J%-1)*PN
8190   E(L)=I%+J%*PN
8200   L=L+1
8210  NEXT J%
8220  S(L)=E(L-1)
8230  E(L)=I%
8240  L=L+1
8250 NEXT I%
8299 RETURN

9000 REM menu
9010 CLS
9020 WHILE M$<>"V" AND M$<>"G" AND M$<>"Q"
9030  LOCATE 14,10
9040  PRINT "Which object do you wish to see "
9050  LOCATE 9,13
9060  PRINT "A VDU (V), wine glass (G), or quit (Q)";
9070  INPUT M$
9080  M$=LEFT$(M$,1)
```

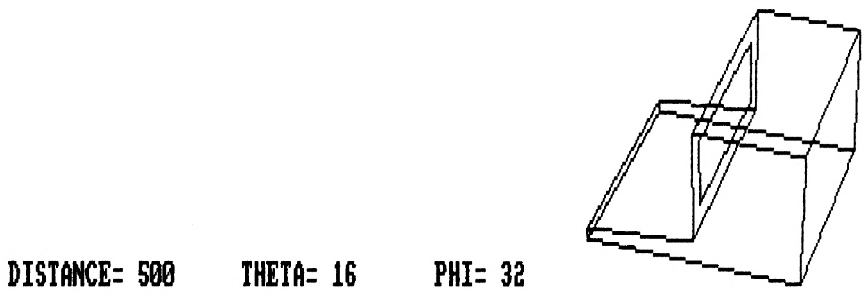
```
9090  M$=UPPER$(M$)
9100 WEND
9110 IF M$="G" THEN M%=1 ELSE M%=0
9120 IF M$="Q" THEN QUIT%=1
9130 MENU%=0
9140 CLG
9150 M$=""
9199 RETURN
```

Once you get the program up and running you can select one of two different demonstration objects – a VDU or a wine glass. The VDU is defined using a complete point and line file, but the wine glass is generated as a solid of rotation from a two-dimensional definition of its profile. Samples of the output of the program can be seen in figs 7.8 and 7.9. You can view either object from various positions by using the C key to move Closer, the F key to move Further away and the arrow keys to move up and down, and left and right, around the object. Each time you press D the object is Drawn from the current position of the observer.

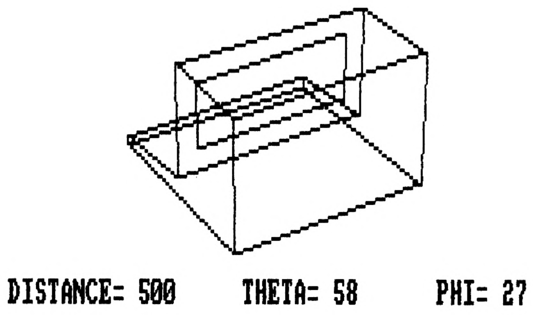
Once you get tired of exploring the example shapes you might like to program in your own shapes by changing the DATA statements, lines 520 to 890 for the VDU and lines 1110 to 1230 for the wine glass. These record the co-ordinates of P points and L lines in the format described in detail earlier but, put simply, the first P DATA statements are the co-ordinates of the points that make up the object and the next L DATA statements are the start and end point of the lines that make up the object. Thus, line 680 states that the first line in the object connects point 0 with point 1 (i.e. the point defined by line 520 with the point defined by line 530). If you want to input your own shape then I would suggest that you draw the shape on graph paper before you start typing DATA statements.

You might also like to change the centre of projection defined by lines 1880 and 1890 although the only value worth changing is ZC which alters the degree of perspective. Its current setting at –1000 produces a slightly exaggerated perspective, but this is useful for demonstration. You can also change the centre of attention defined in line 995 for the VDU and line 1350 for the wine glass.

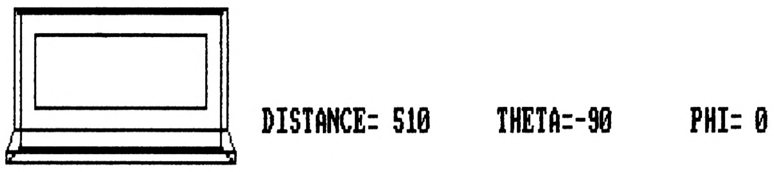
The structure of the program can be seen in the subroutine function table given below. The 'number crunching' subroutines are not optimised for speed but are organised to show the steps in the calculation. Finally it is worth saying that if you have any trouble with the program you should check that you have entered subroutines 5000, 6000 and 7000 correctly as any slight error in the complex calculation will result in rubbish.



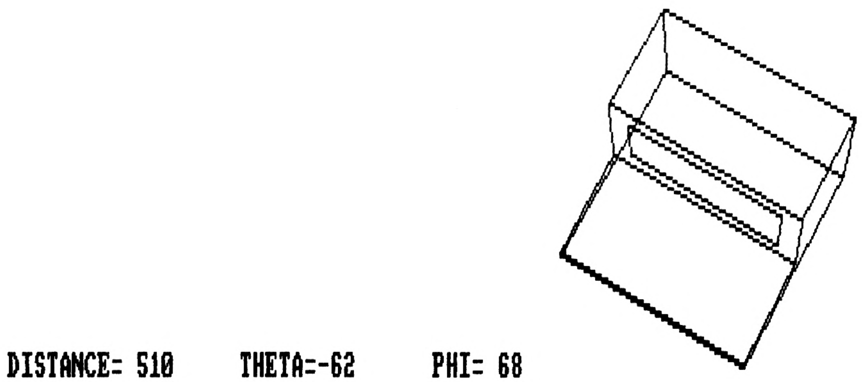
```
DISTANCE= 500     THETA= 16     PHI= 32
```



```
DISTANCE= 500     THETA= 58     PHI= 27
```

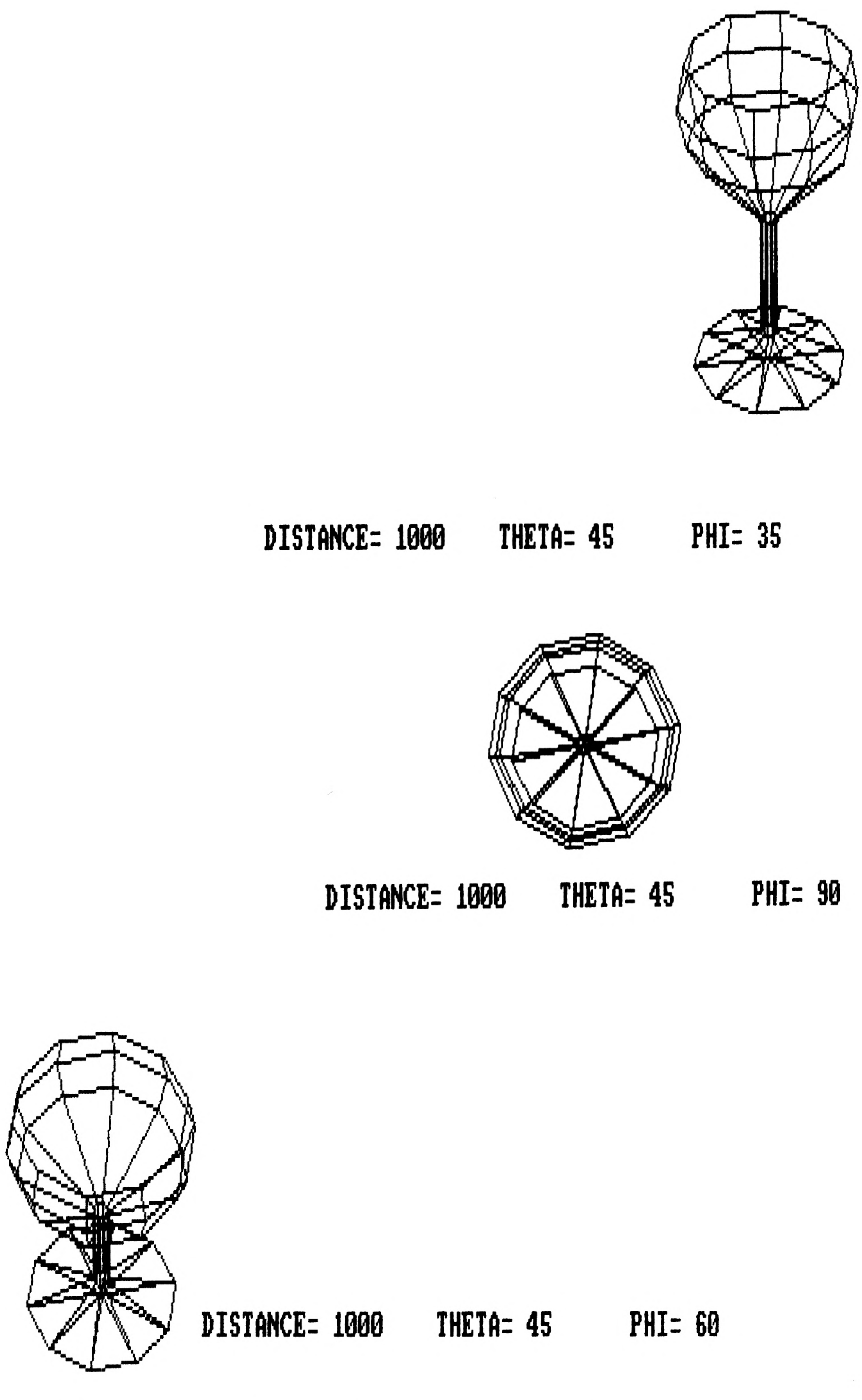


```
DISTANCE= 510     THETA=-90     PHI= 0
```



```
DISTANCE= 510     THETA=-62     PHI= 68
```

Fig 7.8 Views of a VDU using AMVIEW – a sequence of four different views



```
DISTANCE= 1000     THETA= 45       PHI= 35

DISTANCE= 1000     THETA= 45       PHI= 90

DISTANCE= 1000     THETA= 45       PHI= 60
```

Fig 7.9 Views of a wine glass using AMVIEW – a sequence of three views

Subroutine Use – AMVIEW

Subroutine	Function
500	Data for VDU shape
1000	Profile data for wine glass
1800	Initialisation – XC,YC,ZC is the centre of projection
2000	Position input subroutine
3000	Perspective projection routine
4000	Line drawing routine
5000	Calculates SIN, COS and other values for the current view transformation
6000	applies the view transformation to the point file
7000	applies a final 'head up' transformation to make sure that the object is the right way up and centres the object in the Amstrad's screen
8000	Rotates the profile data to produce a full 3D object N copies of the profile are made
9000	Presents menu

AMVIEW is a prototype of programs that have valuable practical application in many areas – from architecture and design to medicine and nuclear technology. Using it is both fun and instructive and it serves to demonstrate that, although a home computer, the Amstrad has a great deal of potential.

Appendix I A resumé of graphics commands

The 464 graphics commands

General

1 In each mode there are a given number of logical colours
Mode 0 – 16 Mode 1 – 4 Mode 2 – 2
2 There are 27 physical colours and these are assigned to logical colours by
INK L, P
where L is the logical colour and P is the physical colour or
INK L, P1, P2
where the assigned colour flashes between P1 and P2
3 The border colour is set by
BORDER P
or
BORDER P1, P2
for a flashing colour

Text

4 The character co-ordinate system changes with mode
Mode 0 – 20 by 25 Mode 1 – 40 by 25 Mode 2 – 80 by 25
5 The text cursor is positioned by LOCATE X,Y
6 The text foreground colour is set by PEN L
7 The text background colour is set by PAPER L
8 User-defined characters are defined by
SYMBOL ascii code, 8 pixel definitions

9 Space for user-defined characters is reserved by
SYMBOL AFTER ascii code
The default is SYMBOL AFTER 240
10 A text window is set by
WINDOW #S,L,R,T,B
11 A text window is cleared by CLS #S
12 The current text cursor position is given by the pair of functions
POS,VPOS

High resolution
13 All high resolution commands treat the screen as if it were 640 by 400 and the origin 0,0 is in the bottom left hand corner
14 Graphics background colour is set by CLG L and graphics foreground colour is set within each graphics command
15 The graphics cursor is moved and pixels plotted by

MOVE X,Y	moves graphics cursor to X,Y
MOVER DX,DY	moves graphics cursor to xg+DX,yg+DY
PLOT X,Y,L	sets pixel at X,Y to colour L
PLOTR DX,DY,L	sets pixel at xg+DX,yg+DY to L
DRAW X,Y,L	draws lines from xg,yg to X,Y in colour L
DRAWR DX,DY,L	draws line from xg,yg to xg+DX,yg+DY in colour L

where xg,yg is the current position of the graphics cursor and L is optional
If L is specified then the graphics foreground colour is set to L and if L is omitted then the current graphics foreground colour is used to plot the pixel or draw the line
16 A graphics window can be defined by
ORIGIN X,Y,L,R,T,B with the new origin at X,Y
17 The graphics window can be cleared by
CLG L which also sets the current graphics background colour to L
18 The current graphics cursor position is given by the pair of functions XPOS,YPOS
19 The logical colour of the pixel at X,Y is returned by TEST X,Y and TESTR DX,DY returns the logical colour of the pixel at xg+DX,yg+DY
20 Text is printed at the graphics cursor position following TAG and at the text cursor following TAGOFF

CPC 664 commands

21 The graphics foreground and background colours can be set

using the GRAPHICS PEN and GRAPHICS PAPER commands
22 The MOVE and MOVER command can now specify a foreground colour
23 All graphics commands can be used with an extra parameter that sets the graphics ink mode
24 The FILL command can be used to fill irregular areas
25 The COPYCHR$ command can be used to find out what character is already printed on the screen
26 The FRAME command can be used for synchronising animation to the TV frame rate
27 MASK can be used to draw dotted lines and suppress the drawing of line endpoints

Index

WORKING GRAPHICS ON THE AMSTRAD CPC464 & 664

Graphics are becoming more and more important in personal computer applications, and whether your interest is in games, serious uses in the home or business applications, it's important to present the screen displays of your programs in the most effective way. This book sets out to show you how to use and how to get the best from the graphics on the Amstrad CPC464 and 664 computers. Subjects covered in detail include animation and sprites, computer-assisted painting, two- and three-dimensional graphics, and charts and graphs.

The emphasis throughout the book is on the practical aspects of Amstrad graphics — you can use the listings as they stand, in which case you will find that the painting program, the three-dimensional viewer and the graphic design program would each cost more than this book if bought as a separate software package. Alternatively, since all the routines are fully explained, you can modify them to suit your own particular application, or use them as a model for your own totally new graphics programs.

All the listings in the book are taken from printer dumps of working programs, so you can be sure no typesetting errors have crept into them; similarly, all the line drawings are taken from working programs, so you will be able to reproduce all the graphics pictures in the book simply by typing in the programs from the listings provided.

All in all, a book for every Amstrad owner whether games player or serious user.

ISBN 0-85242-874-X

9 780852 428740

WORKING GRAPHICS ON THE AMSTRAD CPC464 & 664

ARGUS BOOKS